Rajalakshmi Engineering College

Name: NEIL DANIEL A
Email: 240701356@rajalakshmi.edu.in
Roll no: 240701356
Phone: 8925059757
Branch: REC
Department: I CSE FD
Batch: 2028
Degree: B.E - CSE

Scan to verify results

NeoColab_REC_CS23231_DATA STRUCTURES

REC_DS using C_Week 1_PAH_modified

Attempt : 1
Total Mark : 5
Marks Obtained : 4.25

Section 1 : Coding

1. Problem Statement

Emily is developing a program to manage a singly linked list. The program should allow users to perform various operations on the linked list, such as inserting elements at the beginning or end, deleting elements from the beginning or end, inserting before or after a specific value, and deleting elements before or after a specific value. After each operation, the updated linked list should be displayed.

Your task is to help Emily in implementing the same.

Input Format

The first line contains an integer choice, representing the operation to perform:

- For choice 1 to create the linked list. The next lines contain space-separated

integers, with -1 indicating the end of input.
- For choice 2 to display the linked list.
- For choice 3 to insert a node at the beginning. The next line contains an integer data representing the value to insert.
- For choice 4 to insert a node at the end. The next line contains an integer data representing the value to insert.
- For choice 5 to insert a node before a specific value. The next line contains two integers: value (existing node value) and data (value to insert).
- For choice 6 to insert a node after a specific value. The next line contains two integers: value (existing node value) and data (value to insert).
- For choice 7 to delete a node from the beginning.
- For choice 8 to delete a node from the end.
- For choice 9 to delete a node before a specific value. The next line contains an integer value representing the node before which deletion occurs.
- For choice 10 to delete a node after a specific value. The next line contains an integer value representing the node after which deletion occurs.
- For choice 11 to exit the program.

Output Format

For choice 1, print "LINKED LIST CREATED".

For choice 2, print the linked list as space-separated integers on a single line. If the list is empty, print "The list is empty".

For choice 3, 4, 5, and 6, print the updated linked list with a message indicating the insertion operation.

For choice 7, 8, 9, and 10, print the updated linked list with a message indicating the deletion operation.

For any operation that is not possible print an appropriate error message such as "Value not found in the list".

For choice 11 terminate the program.

For any invalid option, print "Invalid option! Please try again".

Refer to the sample output for formatting specifications.

Sample Test Case

Input: 1
5
3
7
-1
2
11

Output: LINKED LIST CREATED
5 3 7

Answer

```
// You are using GCC
#include <stdio.h>
#include <stdlib.h>

struct node {
    int d;
    struct node *n;
};

void print(struct node** l);

struct node* cN(int d) {
    struct node* a = (struct node*) malloc(sizeof(struct node));
    // if (a == NULL) {
    //     fprintf(stderr, "Memory allocation failed\n");
    //     exit(EXIT_FAILURE);
    // }
    a->d = d;
    a->n = NULL;
    return a;
}

void insertAtEnd(struct node** l, int d) {
    struct node* a= cN(d);
    if (a == NULL) {
        return;
    }
    if (*l == NULL) {
        *l = a;
```

```
    }else{
        struct node* c = *l;
        while (c->n != NULL) {
            c=c->n;
        }
        c->n = a;
    }
}

void insertAtBegin(struct node** l, int d) {
    struct node* a= cN(d);
    if (*l == NULL) {
        *l = a;
    }else{
        struct node* c = *l;
        a->n = c;
        *l = a;
    }
}

// CHECK
void insertAtBefore(struct node** l, int d, int v) {
    struct node* a= cN(d);
    if (a == NULL) {
        return;
    }
    if (*l == NULL) {
        *l = a;
    }else{
        struct node* c = *l;
        if (c->d == v) {
            a->n = *l;
            *l = a;
            return;
        }else{
            while (c->n != NULL && c->n->d != v) {
                c=c->n;
            }
            if (c->n == NULL) {
                printf("Value not found in the list\n");
                return;
            }
```

```
            a->n = c->n;
            c->n = a;
        }
    }
}

void insertAtAfter(struct node** l, int d, int v) {
    struct node* a= cN(d);
    if (a == NULL) {
        return;
    }
    if (*l == NULL) {
        *l = a;
    }else{
        struct node* c = *l;
        if (c->d == v) {
            a->n = c->n;
            c->n = a;
        }else{
            while (c != NULL && c->d != v) {
                c=c->n;
            }
            if (c == NULL) {
                printf("Value not found in the list\n");
                return;
            }
            a->n = c->n;
            c->n = a;
        }
    }
}

void deleteAfter(struct node** l, int v) {
    if (*l != NULL) {
        struct node* c = *l;
        while (c != NULL && c->d != v) {
            if (c == NULL) {
                printf("Operation not possible\n");
                return;
            }
            c=c->n;
        }
```

```
        if (c == NULL || c->n == NULL) {
            printf("Operation not possible\n");
            return;
        }
        struct node* a = c->n;
        c->n = a->n;
        free(a);
    }else {
        printf("Operation not possible\n");
        return;
    }
}

void deleteBefore(struct node** l, int v) {
    if (*l != NULL) {
        struct node* c = *l;
        if (c->n != NULL && c->n->d == v) {
                printf("No node exists before the value\n");
            return;
        }
        if (c != NULL && c->d == v) {
                printf("No node exists before the value\n");
            return;
        }
        if (c->n->n != NULL && c->n->n->d == v) {
            struct node* a = c;
            *l = a->n;
            free(a);
            printf("The linked list after deletion before a value is:\n");
            print(l);
            return;
        }
        while (c->n != NULL && c->n->d != v) {
            if (c == NULL) {
                printf("Operation not possible\n");
                return;
            }
            c=c->n;
        }
        if (c->n == NULL) {
                printf("No node exists before the value\n");
            return;
```

```
        }
        struct node* a = c->n;
        c->n = a->n;
        free(a);
    }else {
        printf("Operation not possible\n");
        return;
    }
    printf("The linked list after deletion before a value is:\n");
    print(l);
}


int deleteFirst(struct node** l) {
    if (*l == NULL) {
        printf("List is empty\n");
        return 0;
    }
    struct node* a= *l;
    *l = a->n;
    free(a);
    return 1;
}


int deleteEnd(struct node** l) {
    if (*l == NULL) {
        printf("List is empty\n");
        return 0;
    }
    struct node* c= *l;
    while (c->n->n != NULL) {
        c=c->n;
    }
    c->n = NULL;
    return 1;
}


void print(struct node** l) {
    if (*l == NULL) {
        printf("The list is empty\n");
```

```
        return;
    }
    struct node* c = *l;
    while (c != NULL) {
        printf("%d ", c->d);
        c=c->n;
    }
    printf("\n");
}

int main() {
    struct node* l = NULL;
    int t, v;
    while (1) {
        int c;
        scanf("%d", &c);

        switch (c) {
            case 1:
                scanf("%d", &t);
                while (t != -1) {
                    insertAtEnd(&l, t);
                    scanf("%d", &t);
                }
                printf("LINKED LIST CREATED\n");
                break;
            case 2:
                print(&l);
                break;
            case 3:
                scanf("%d", &t);
                insertAtBegin(&l, t);
                printf("The linked list after insertion at the beginning is:\n");
                print(&l);
                break;
            case 4:
                scanf("%d", &t);
                insertAtEnd(&l, t);
                printf("The linked list after insertion at the end is:\n");
                print(&l);
                break;
            case 5:
```

```
            scanf("%d", &v);
            scanf("%d", &t);
            insertAtBefore(&l, t, v);
            printf("The linked list after insertion before a value is:\n");
            print(&l);
            break;
        case 6:
            scanf("%d", &v);
            scanf("%d", &t);
            insertAtAfter(&l, t, v);
            printf("The linked list after insertion after a value is:\n");
            print(&l);
            break;
        case 7:
            if (deleteFirst(&l) == 1) {
                printf("The linked list after deletion from the beginning is:\n");
                print(&l);
            }
            break;
        case 8:
            if (deleteEnd(&l) == 1) {
                printf("The linked list after deletion from the end is:\n");
                print(&l);
            }
            break;
        case 9:
            scanf("%d", &v);
            deleteBefore(&l, v);
            break;
        case 10:
            scanf("%d", &v);
            deleteAfter(&l, v);
            printf("The linked list after deletion after a value is:\n");
            print(&l);
            break;



        case 11:
            return 0;

        default:
```

```
            printf("Invalid option! Please try again\n");
        }
    }
}
```

Status : Partially correct ***Marks : 0.75/1***

2. Problem Statement

Bharath is very good at numbers. As he is piled up with many works, he decides to develop programs for a few concepts to simplify his work. As a first step, he tries to arrange even and odd numbers using a linked list. He stores his values in a singly-linked list.

Now he has to write a program such that all the even numbers appear before the odd numbers. Finally, the list is printed in such a way that all even numbers come before odd numbers. Additionally, the even numbers should be in reverse order, while the odd numbers should maintain their original order.

Example

Input:

6

3 1 0 4 30 12

Output:

12 30 4 0 3 1

Explanation:

Even elements: 0 4 30 12

Reversed Even elements: 12 30 4 0

Odd elements: 3 1

So the final list becomes: 12 30 4 0 3 1

Input Format

The first line consists of an integer n representing the size of the linked list.

The second line consists of n integers representing the elements separated by space.

Output Format

The output prints the rearranged list separated by a space.

The list is printed in such a way that all even numbers come before odd numbers and the even numbers should be in reverse order, while the odd numbers should maintain their original order.

Refer to the sample output for the formatting specifications.

Sample Test Case

Input: 6
3 1 0 4 30 12

Output: 12 30 4 0 3 1

Answer

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node{
   int data;
   struct node *next;
}node;
node* create(int val){
   node* newnode=(node*)malloc(sizeof(node));
   newnode->data=val;
   newnode->next=NULL;
   return newnode;
}
void insert(node** head,int val){
   node* newnode=create(val);
   if(val%2==0){
      newnode->next=*head;
      *head=newnode;
   }else{
      if(*head==NULL){
```

```
            *head=newnode;
        }else{
            node*temp=*head;
            while(temp->next!=NULL){
                temp=temp->next;
            }
            temp->next=newnode;
        }
    }
}
void printlist(node* head){
    node* temp=head;
    while(temp!=NULL){
        printf("%d ",temp->data);
        temp=temp->next;
    }
    printf("\n");
}

int main(){
    int n,val;
    node* head=NULL;
    scanf("%d",&n);
    for(int i=0;i<n;i++){
        scanf("%d",&val);
        insert(&head,val);
    }
    printlist(head);
    return 0;
}
```

Status : Correct

Marks : 1/1

3. Problem Statement

John is working on evaluating polynomials for his math project. He needs to compute the value of a polynomial at a specific point using a singly linked list representation.

Help John by writing a program that takes a polynomial and a value of x as input, and then outputs the computed value of the polynomial.

Example

Input:

2

13

12

11

1

Output:

36

Explanation:

The degree of the polynomial is 2.

Calculate the value of x2: 13 * 12 = 13.

Calculate the value of x1: 12 * 11 = 12.

Calculate the value of x0: 11 * 10 = 11.

Add the values of x2, x1 and x0 together: 13 + 12 + 11 = 36.

Input Format

The first line of input consists of the degree of the polynomial.

The second line consists of the coefficient x2.

The third line consists of the coefficient of x1.

The fourth line consists of the coefficient x0.

The fifth line consists of the value of x, at which the polynomial should be evaluated.

Output Format

The output is the integer value obtained by evaluating the polynomial at the given value of x.

Refer to the sample output for formatting specifications.

Sample Test Case

Input: 2
13
12
11
1

Output: 36

Answer

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

typedef struct Node {
    int coefficient;
    struct Node* next;
} Node;

typedef struct Polynomial {
    Node* head;
} Polynomial;

Node* createNode(int coefficient) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->coefficient = coefficient;
    newNode->next = NULL;
    return newNode;
}
```

```
void addCoefficient(Polynomial* poly, int coefficient) {
    Node* newNode = createNode(coefficient);
    if (poly->head == NULL) {
        poly->head = newNode;
    } else {
        Node* current = poly->head;
        while (current->next != NULL)
            current = current->next;
        current->next = newNode;
    }
}

int power(int base, int exp) {
    int result = 1;
    for (int i = 0; i < exp; i++)
        result *= base;
    return result;
}

int evaluatePolynomial(Polynomial* poly, int degree, int x) {
    int result = 0;
    int index = 0;
    Node* current = poly->head;
    while (current != NULL) {
        int exponent = degree - index;
        result += current->coefficient * power(x, exponent);
        index++;
        current = current->next;
    }
    return result;
}

int main() {
    Polynomial poly;
    poly.head = NULL;

    int degree;
    scanf("%d", &degree);

    for (int i = 0; i <= degree; i++) {
        int coeff;
```

```
        scanf("%d", &coeff);
        addCoefficient(&poly, coeff);
    }

    int x;
    scanf("%d", &x);

    int result = evaluatePolynomial(&poly, degree, x);
    printf("%d\n", result);

    Node* current = poly.head;
    while (current != NULL) {
        Node* temp = current;
        current = current->next;
        free(temp);
    }

    return 0;
}
```

Status : Correct ***Marks : 1/1***

4. Problem Statement

Imagine you are managing the backend of an e-commerce platform. Customers place orders at different times, and the orders are stored in two separate linked lists. The first list holds the orders from morning, and the second list holds the orders from the evening.

Your task is to merge the two lists so that the final list holds all orders in sequence from the morning list followed by the evening orders, in the same order

Input Format

The first line contains an integer n , representing the number of orders in the morning list.

The second line contains n space-separated integers representing the morning orders.

The third line contains an integer m , representing the number of orders in the evening list.

The fourth line contains m space-separated integers representing the evening orders.

Output Format

The output should be a single line containing space-separated integers representing the merged order list, with morning orders followed by evening orders.

Refer to the sample output for formatting specifications.

Sample Test Case

Input: 3
101 102 103
2
104 105

Output: 101 102 103 104 105

Answer

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node{
   int data;
   struct node* next;
}node;
node* create(int val){
   node* newnode=(node*)malloc(sizeof(node));
   newnode->data=val;
   newnode->next=NULL;
   return newnode;
}
void insert(node** head,int val){
   node* newnode=create(val);
   if(*head==NULL){
      *head=newnode;
   }else{
      node* temp=*head;
```

```
        while(temp->next!=NULL){
            temp=temp->next;
        }
        temp->next=newnode;
    }
}
node* merge(node* head1,node* head2){
    if(head1==NULL) return head2;
    if(head2==NULL) return head1;
    node* temp=head1;
    while(temp->next!=NULL){
        temp=temp->next;
    }
    temp->next=head2;
    return head1;
}
void printlist(node* head){
    node* temp=head;
    while(temp!=NULL){
        printf("%d ",temp->data);
        temp=temp->next;
    }
}
int main(){
    int n,m,val;
    node* head1=NULL;
    node* head2=NULL;
    scanf("%d",&n);
    for(int i=0;i<n;i++){
        scanf("%d",&val);
        insert(&head1,val);
    }
    scanf("%d",&m);
    for(int i=0;i<m;i++){
        scanf("%d",&val);
        insert(&head2,val);
    }
    node* merged=merge(head1,head2);
    printlist(merged);
    return 0;
}
```

Status : Correct ***Marks : 1/1***

5. Problem Statement

Write a program to manage a singly linked list. The program should allow users to perform various operations on the linked list, such as inserting elements at the beginning or end, deleting elements from the beginning or end, inserting before or after a specific value, and deleting elements before or after a specific value. After each operation, the updated linked list should be displayed.

Input Format

The first line contains an integer choice, representing the operation to perform:

- For choice 1 to create the linked list. The next lines contain space-separated integers, with -1 indicating the end of input.
- For choice 2 to display the linked list.
- For choice 3 to insert a node at the beginning. The next line contains an integer data representing the value to insert.
- For choice 4 to insert a node at the end. The next line contains an integer data representing the value to insert.
- For choice 5 to insert a node before a specific value. The next line contains two integers: value (existing node value) and data (value to insert).
- For choice 6 to insert a node after a specific value. The next line contains two

integers: value (existing node value) and data (value to insert).
- For choice 7 to delete a node from the beginning.
- For choice 8 to delete a node from the end.
- For choice 9 to delete a node before a specific value. The next line contains an integer value representing the node before which deletion occurs.
- For choice 10 to delete a node after a specific value. The next line contains an integer value representing the node after which deletion occurs.
- For choice 11 to exit the program.

Output Format

For choice 1, print "LINKED LIST CREATED".

For choice 2, print the linked list as space-separated integers on a single line. If the list is empty, print "The list is empty".

For choice 3, 4, 5, and 6, print the updated linked list with a message indicating the insertion operation.

For choice 7, 8, 9, and 10, print the updated linked list with a message indicating the deletion operation.

For any operation that is not possible print an appropriate error message such as "Value not found in the list".

For choice 11 terminate the program.

For any invalid option, print "Invalid option! Please try again".

Refer to the sample output for formatting specifications.

Sample Test Case

Input: 1
5
3
7
-1
2
11
Output: LINKED LIST CREATED

5 3 7

Answer

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

// Define the node structure
struct node {
   int d;            // Data part of the node
   struct node *n;    // Pointer to the next node
};

// Function prototype for printing the list
void print(struct node** l);

// Function to create a new node
struct node* cN(int d) {
   // Allocate memory for the new node
   struct node* a = (struct node*) malloc(sizeof(struct node));
   // Check if memory allocation was successful
   if (a == NULL) {
      fprintf(stderr, "Memory allocation failed\n");
      exit(EXIT_FAILURE); // Exit if allocation fails
   }
   // Initialize the node's data and next pointer
   a->d = d;
   a->n = NULL;
   return a; // Return the newly created node
}

// Function to insert a node at the end of the list
void insertAtEnd(struct node** l, int d) {
   struct node* a = cN(d); // Create the new node
   // If the list is empty, the new node becomes the head
   if (*l == NULL) {
      *l = a;
   } else {
      // Traverse to the last node
      struct node* c = *l;
      while (c->n != NULL) {
         c = c->n;
      }
```

```
        // Link the last node to the new node
        c->n = a;
    }
}

// Function to insert a node at the beginning of the list
void insertAtBegin(struct node** l, int d) {
    struct node* a = cN(d); // Create the new node
    // Make the new node point to the current head
    a->n = *l;
    // Update the head to be the new node
    *l = a;
}

// Function to insert a node before a specific value
void insertAtBefore(struct node** l, int d, int v) {
    // If the list is empty, cannot insert before anything specific
    if (*l == NULL) {
        printf("Value not found in the list\n");
        return;
    }
    // Check if the value is at the head
    if ((*l)->d == v) {
        insertAtBegin(l, d); // Insert at the beginning
        return;
    }

    struct node* c = *l;
    // Traverse to find the node *before* the one with value v
    while (c->n != NULL && c->n->d != v) {
        c = c->n;
    }

    // If the value was not found (or c->n is NULL)
    if (c->n == NULL) {
        printf("Value not found in the list\n");
        return;
    }

    // Value found, insert the new node
    struct node* a = cN(d);
    a->n = c->n;
```

```
    c->n = a;
}

// Function to insert a node after a specific value
void insertAtAfter(struct node** l, int d, int v) {
    struct node* c = *l;
    // Traverse to find the node with value v
    while (c != NULL && c->d != v) {
        c = c->n;
    }

    // If the value was not found
    if (c == NULL) {
        printf("Value not found in the list\n");
        return;
    }

    // Value found, insert the new node after it
    struct node* a = cN(d);
    a->n = c->n;
    c->n = a;
}

// Function to delete the first node
int deleteFirst(struct node** l) {
    // If the list is empty, cannot delete
    if (*l == NULL) {
        printf("List is empty\n");
        return 0; // Indicate failure
    }
    // Store the current head
    struct node* a = *l;
    // Update the head to the next node
    *l = a->n;
    // Free the old head
    free(a);
    return 1; // Indicate success
}

// Function to delete the last node
int deleteEnd(struct node** l) {
    // If the list is empty
```

```
    if (*l == NULL) {
        printf("List is empty\n");
        return 0; // Indicate failure
    }
    // If there is only one node
    if ((*l)->n == NULL) {
        free(*l);
        *l = NULL;
        return 1; // Indicate success
    }

    // Traverse to the second to last node
    struct node* c = *l;
    while (c->n->n != NULL) {
        c = c->n;
    }
    // Free the last node
    free(c->n);
    // Set the second to last node's next pointer to NULL
    c->n = NULL;
    return 1; // Indicate success
}


// Function to delete the node before a specific value
int deleteBefore(struct node** l, int v) {
    // Cannot delete before if list is empty or has only one node
    if (*l == NULL || (*l)->n == NULL) {
        printf("Operation not possible\n");
        return 0;
    }

    // Check if the node to delete is the head (i.e., v is in the second node)
    if ((*l)->n->d == v) {
        struct node* temp = *l;
        *l = (*l)->n;
        free(temp);
        return 1; // Indicate success
    }

    struct node* c = *l;
    // Traverse until c->n->n contains the value v
```

```
    while (c->n != NULL && c->n->n != NULL) {
        if (c->n->n->d == v) {
            // Node to delete is c->n
            struct node* temp = c->n;
            c->n = c->n->n; // Bypass the node to be deleted
            free(temp);
            return 1; // Indicate success
        }
        c = c->n;
    }

    // If the loop finishes without finding the condition
    printf("Operation not possible\n");
    return 0; // Indicate failure (value not found or structure doesn't allow deletion
before)
}


// Function to delete the node after a specific value
int deleteAfter(struct node** l, int v) {
    struct node* c = *l;
    // Traverse to find the node with value v
    while (c != NULL && c->d != v) {
        c = c->n;
    }

    // If value v is not found, or if it's the last node
    if (c == NULL || c->n == NULL) {
        printf("Operation not possible\n");
        return 0; // Indicate failure
    }

    // Node c contains v, and c->n exists (the node to delete)
    struct node* a = c->n; // Node to delete
    c->n = a->n;          // Bypass the node to be deleted
    free(a);              // Free the deleted node
    return 1; // Indicate success
}


// Function to print the linked list
void print(struct node** l) {
```

```
    // If the list is empty
    if (*l == NULL) {
        printf("The list is empty\n");
        return;
    }
    // Traverse and print each node's data
    struct node* c = *l;
    while (c != NULL) {
        printf("%d", c->d);
        if (c->n != NULL) {
            printf(" "); // Print space between elements
        }
        c = c->n;
    }
    printf("\n"); // Print newline at the end
}

// Main function to drive the program
int main() {
    struct node* l = NULL; // Initialize the head pointer to NULL
    int t, v;           // Temporary variables for input data and value
    int c;              // Variable for user choice

    // Loop indefinitely until choice 11 is entered
    while (1) {
        scanf("%d", &c); // Read the user's choice

        switch (c) {
            case 1: // Create linked list
                scanf("%d", &t);
                while (t != -1) { // Read until -1 sentinel value
                    insertAtEnd(&l, t);
                    scanf("%d", &t);
                }
                printf("LINKED LIST CREATED\n");
                break;
            case 2: // Display list
                print(&l);
                break;
            case 3: // Insert at beginning
                scanf("%d", &t);
                insertAtBegin(&l, t);
```

```
        printf("The linked list after insertion at the beginning is:\n");
        print(&l);
        break;
    case 4: // Insert at end
        scanf("%d", &t);
        insertAtEnd(&l, t);
        printf("The linked list after insertion at the end is:\n");
        print(&l);
        break;
    case 5: // Insert before value
        scanf("%d", &v); // Value to insert before
        scanf("%d", &t); // Data to insert
        insertAtBefore(&l, t, v);
        // Message printed within function if value not found
        // Otherwise print the updated list here
        printf("The linked list after insertion before a value is:\n");
        print(&l);
        break;
    case 6: // Insert after value
        scanf("%d", &v); // Value to insert after
        scanf("%d", &t); // Data to insert
        insertAtAfter(&l, t, v);
        // Message printed within function if value not found
        // Otherwise print the updated list here
        printf("The linked list after insertion after a value is:\n");
        print(&l);
        break;
    case 7: // Delete from beginning
        if (deleteFirst(&l) == 1) { // Check if deletion was successful
            printf("The linked list after deletion from the beginning is:\n");
            print(&l);
        }
        // Error message printed within function if list was empty
        break;
    case 8: // Delete from end
        if (deleteEnd(&l) == 1) { // Check if deletion was successful
            printf("The linked list after deletion from the end is:\n");
            print(&l);
        }
         // Error message printed within function if list was empty
        break;
    case 9: // Delete before value
```

```
            scanf("%d", &v);
            if (deleteBefore(&l, v) == 1) { // Check if deletion was successful
                printf("The linked list after deletion before a value is:\n");
                print(&l);
            }
            // Error message printed within function if operation not possible
            break;
        case 10: // Delete after value
            scanf("%d", &v);
            if (deleteAfter(&l, v) == 1) { // Check if deletion was successful
                printf("The linked list after deletion after a value is:\n");
                print(&l);
            }
            // Error message printed within function if operation not possible
            break;
        case 11: // Exit
            // Clean up allocated memory before exiting (optional but good
practice)
            // while (deleteFirst(&l)); // Repeatedly delete first node until list is
empty
            return 0; // Terminate the program
        default: // Invalid option
            printf("Invalid option! Please try again\n");
    }
  }
  // The program should not reach here due to the infinite loop and return in
case 11
  return 0;
}
```

Status : Partially correct **Marks : 0.5/1**