Algorithm: Implement Stack Using Linked List

1. **Initialize Stack:**
   - Start with top = NULL (stack is empty).
2. **Push Operation (Insert an Element):**
   - Allocate memory for a new node (newnode).
   - If memory allocation fails, print "Memory allocation failed" and exit.
   - Input data to be pushed.
   - Set newnode->data = input data.
   - Set newnode->next = top.
   - Update top = newnode.
3. **Pop Operation (Remove an Element):**
   - If top == NULL, print "Stack Underflow" (stack is empty), and exit.
   - Store the data of top node in a variable to return later.
   - Set a temporary pointer temp = top.
   - Update top = top->next.
   - Free the memory of temp.
   - Return or print the popped data.
4. **Peek Operation (View Top Element):**
   - If top == NULL, print "Stack is empty".
   - Else, print the data in top node.
5. **Check if Stack is Empty:**
   - If top == NULL, stack is empty.
   - Else, stack is not empty.
6. **End:**
   - The stack operations are performed using linked list nodes with top pointing to the current top node.

Program:

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

// Define the structure for a node
struct Node {
    int data;
    struct Node* next;
};

// Initialize top pointer
struct Node* top = NULL;

// Function to push an element onto the stack
void push(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        return;
    }
    newNode->data = value;
    newNode->next = top;
    top = newNode;
    printf("%d pushed onto the stack\n", value);
}

// Function to pop an element from the stack
void pop() {
    struct Node* temp;
    if (top == NULL) {
```

```c
        printf("Stack Underflow\n");
        return;
    }
     temp= top;
    printf("%d popped from the stack\n", top->data);
    top = top->next;
    free(temp);
}

// Function to peek at the top element of the stack
void peek() {
    if (top == NULL) {
        printf("Stack is empty\n");
        return;
    }
    printf("Top element is %d\n", top->data);
}

// Function to display the elements of the stack
void display() {
    struct Node* temp;
    if (top == NULL) {
        printf("Stack is empty\n");
        return;
    }
    temp = top;
    printf("Stack elements: ");
    while (temp) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
// Main function to demonstrate stack operations
int main() {
    push(10);
    push(20);
    push(30);
    display();   // Optional: to see the stack after pushes
    pop();
    display();   // Optional: to see the stack after pop
    return 0;
}
```

Algorithm: Implement Queue Using Linked List

1. **Initialize Queue:**
   - Start with front = NULL and rear = NULL (queue is empty).
2. **Enqueue Operation (Insert an Element):**
   - Allocate memory for a new node (newnode).
   - If memory allocation fails, print "Memory allocation failed" and exit.
   - Input data to be enqueued.
   - Set newnode->data = input data.
   - Set newnode->next = NULL.
   - If rear == NULL (queue is empty):
     - Set front = rear = newnode.
   - Else:
     - Set rear->next = newnode.
     - Update rear = newnode.
3. **Dequeue Operation (Remove an Element):**
   - If front == NULL, print "Queue Underflow" (queue is empty), and exit.
   - Store the data of front node in a variable to return later.
   - Set a temporary pointer temp = front.
   - Update front = front->next.
   - If front == NULL after update, set rear = NULL.
   - Free the memory of temp.
   - Return or print the dequeued data.
4. **Peek Operation (View Front Element):**
   - If front == NULL, print "Queue is empty".
   - Else, print the data in front node.
5. **Check if Queue is Empty:**
   - If front == NULL, queue is empty.
   - Else, queue is not empty.
6. **End:**
   - The queue operations are performed using linked list nodes with front pointing to the first node and rear pointing to the last node.

Program:

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

// Define the structure for a node
struct Node {
  int data;
  struct Node* next;
};

// Initialize front and rear pointers
struct Node* front = NULL;
struct Node* rear = NULL;

// Function to insert an element into the queue
void enqueue(int value) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  if (!newNode) {
    printf("Memory allocation failed\n");
    return;
  }
  newNode->data = value;
  newNode->next = NULL;
  if (rear == NULL) {
    front = rear = newNode;
  } else {
```

```c
        rear->next = newNode;
        rear = newNode;
    }
    printf("%d enqueued to the queue\n", value);
}

// Function to delete an element from the queue
void dequeue() {
    struct Node* temp;
    if (front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    temp = front;
    front = front->next;
    if (front == NULL)
        rear = NULL;
    printf("%d dequeued from the queue\n", temp->data);
    free(temp);
}

// Function to display the elements of the queue
void display() {
    if (front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct Node* temp = front;
    printf("Queue elements: ");
    while (temp) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Main function to demonstrate queue operations
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    dequeue();
    display();
    return 0;
}
```