Aim: Implement Stack ADT using Linked List.

Theory:

Stack is a linear data structure that follows the Last-In-First-Out (LIFO) order of operations. This means the last element added to the stack will be the first one to be removed. There are different ways using which we can implement stack data structure in C. In this experiment, we will learn how to implement a stack using a linked list in C, its basic operation along with their time and space complexity analysis.

Procedure:

Initialize

Function: Initialize()

Description: Creates an empty stack.

Algorithm:

Set the top pointer to null.

2. Push

Function: Push(value)

Description: Inserts a new element with value onto the top of the stack.

Algorithm:

Create a new node and assign value to it.

Set the next pointer of the new node to point to the current top.

Update top to point to the new node.

3. Pop

Function: Pop()

Description: Removes and returns the element from the top of the stack.

Algorithm:

If top is null, the stack is empty; return an error or null.

Store the value of the top node in a variable (for return).

Update top to point to top.next.

Delete the old top node to free memory.

Return the stored value.

4. Peek

Function: Peek()

Description: Returns the top element of the stack without removing it.

Algorithm:

If top is null, return an error or null.

Return the value of the top node.

5. IsEmpty

Function: IsEmpty()

Description: Checks if the stack is empty.

Algorithm:

If top is null, return true (stack is empty); otherwise, return false.

Summary of the Procedures

Initialize: Set top to null.

Push(value): Create a new node with value, link it to top, and update top.

Pop(): Remove top node, return its value, and update top.

Peek(): Return the value of top without removing it.

IsEmpty(): Return true if top is null, else false.

Code:

```
// C program to implement a stack using linked list
 #include <stdio.h>
 #include <stdlib.h>
 // ________LINKED LIST UTILITY FUNCITON____________
 // Define the structure for a node of the linked list
 typedef struct Node {
int data;
```

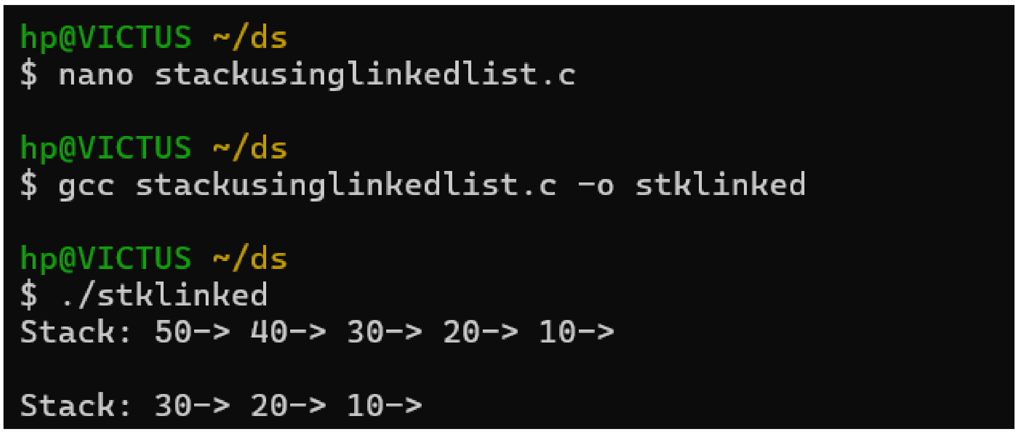
```
struct Node* next;
} node;
// linked list utility function
node* createNode(int data)
{
// allocating memory
node* newNode = (node*)malloc(sizeof(node));
// if memory allocation is failed
if (newNode == NULL)
return NULL;
// putting data in the node
newNode->data = data;
newNode->next = NULL;
return newNode;
}
// fuction to insert data before the head node
int insertBeforeHead(node** head, int data)
{
// creating new node
node* newNode = createNode(data);
// if malloc fail, return error code
if (!newNode)
return-1;
// if the linked list is empty
if (*head == NULL) {
*head = newNode;
return 0;
}
newNode->next = *head;
```

```
*head = newNode;
return 0;
}
// deleting head node
int deleteHead(node** head)
{
// no need to check for empty stack as it is already
// being checked in the caller function
node* temp = *head;
*head = (*head)->next;
free(temp);
return 0;
}
// _________STACK IMPLEMENTATION STARTS HERE_________
// Function to check if the stack is empty or not
int isEmpty(node** stack) { return *stack == NULL; }
// Function to push elements to the stack
void push(node** stack, int data)
{
// inserting the data at the beginning of the linked
// list stack
// if the insertion function returns the non- zero
// value, it is the case of stack overflow
if (insertBeforeHead(stack, data)) {
printf("Stack Overflow!\n");
}
}
// Function to pop an element from the stack
int pop(node** stack)
```

```
{
// checking underflow condition
if (isEmpty(stack)) {
printf("Stack Underflow\n");
return-1;
}
// deleting the head.
deleteHead(stack);
}
// Function to return the topmost element of the stack
int peek(node** stack)
{
// check for empty stack
if (!isEmpty(stack))
return (*stack)->data;
else
return-1;
}
// Function to print the Stack
void printStack(node** stack)
{
node* temp = *stack;
while (temp != NULL) {
printf("%d-> ", temp->data);
temp = temp->next;
}
printf("\n");
}
// driver code
```

```
 int main()
 {
 // Initialize a new stack top pointer
 node* stack = NULL;
 // Push elements into the stack
 push(&stack, 10);
 push(&stack, 20);
 push(&stack, 30);
 push(&stack, 40);
 push(&stack, 50);
 // Print the stack
 printf("Stack: ");
printStack(&stack);
 // Pop elements from the stack
 pop(&stack);
 pop(&stack);
 // Print the stack after deletion of elements
 printf("\nStack: ");
 printStack(&stack);
 return 0;
 }
```

Output:



```
hp@VICTUS ~/ds
$ nano stackusinglinkedlist.c

hp@VICTUS ~/ds
$ gcc stackusinglinkedlist.c -o stklinked

hp@VICTUS ~/ds
$ ./stklinked
Stack: 50-> 40-> 30-> 20-> 10->

Stack: 30-> 20-> 10->
```

Conclusion: This experiment shows how to implement a Stack ADT using a linked list, an alternative to the array-based implementation. It emphasizes the use of linked lists to create a dynamic stack that can grow as needed, providing flexibility and avoiding fixed-size limitations.