LAB SESSION 05

Stack Implementation Using Linked List

THEORY

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. In a linked list implementation of a stack, each element is represented by a node containing:

1. Data – the value stored in the node.
2. Pointer – a reference to the next node in the stack.

Unlike the array-based stack, the linked list implementation does not require a fixed size. Memory is allocated dynamically, and the stack can grow or shrink as needed.

Structure of a Node

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

The **top** pointer always points to the most recently inserted node.

- **push()** → Create a new node and make it the new top.
- **pop()** → Remove the node currently at top and move top to the next node.

Advantages of Linked List Stack

- **Dynamic size** – No fixed memory allocation required.
- No overflow unless the system is out of memory.
- Efficient insertion and deletion at the top.

Disadvantages

- Extra memory for storing pointers.
- Slightly slower access compared to array stack due to pointer dereferencing.

Applications

- Undo operations in text editors
- Function call management
- Parentheses matching
- Reversing strings or data

```
+---------+     +---------+     +---------+
| Data  |     | Data  |     | Data  |
|  30   |     |  20   |     |  10   |
| Next --+----> | Next --+----> | Next --+--> NULL
+---------+     +---------+     +---------+
```

Fig. 5.1. Node Structure

PROCEDURE

1. Define the Node structure containing data and next.
2. Maintain a top pointer to track the top of the stack.

3. Implement push, pop, and display functions.
4. Use dynamic memory allocation for new nodes.
5. Compile and test the program with sample operations.

CODE

```
#include <iostream>
using namespace std;

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

Node* top = NULL;

void push(int val) {
    Node* newNode = new Node();
    newNode->data = val;
    newNode->next = top;
    top = newNode;
    cout << val << " pushed into stack.\n";
}

void pop() {
    if (top == NULL) {
        cout << "Stack Underflow!\n";
        return;
    }
    cout << top->data << " popped from stack.\n";
    Node* temp = top;
    top = top->next;
    delete temp;
}

void display() {
    if (top == NULL) {
        cout << "Stack is empty.\n";
        return;
    }
    cout << "Stack elements: ";
    Node* temp = top;
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
```

```
    push(10);
    push(20);
    push(30);
    display();
    pop();
    display();
    return 0;
}
```

EXPECTED OUTPUT

10 pushed into stack.
20 pushed into stack.
30 pushed into stack.
Stack elements: 30 20 10
30 popped from stack.
Stack elements: 20 10

EXERCISE

1. Modify the program to allow user input for all stack operations.
2. Implement a peek() function for linked list stack.
3. Check if the stack is empty before performing pop or display.
4. Reverse a linked list stack using another stack.
5. Implement balanced parentheses checking using a linked list stack.