LAB 11

CSE225L

Stack (Linked List—Based)

In this lab, we will:

- Design and implement the Stack ADT using a linked list—based structure.
- Create the **StackType** class with methods for Push, Pop, Top, and status checks (IsFull, IsEmpty).
- Test the stack by performing operations such as inserting, removing, and retrieving elements, and handling stack overflow and underflow conditions.
- Implement an infix-to-postfix converter and postfix evaluator using the stack to compute the result of arithmetic expressions.

STACK (LINKED LIST–BASED)

stacktype.h

```
#ifndef STACKTYPE_H
#define STACKTYPE_H

const int SIZE = 5;

// Exception class thrown by Push when the stack is full
class FullStack {};

// Exception class thrown by Pop and Top when the stack is empty
class EmptyStack {};

template <class T>
class StackType
{
    struct Node
    {
        T data;
        Node* next;
    };
private:
    Node* head;
public:
    StackType();
    ~StackType();
    bool IsEmpty();
    bool IsFull();
    void Push(T);
    void Pop();
    void Diagnose(); // Optional
    T Top();
};

#endif // STACKTYPE_H
```

stacktype.cpp

```
#include <iostream>
#include "stacktype.h"
using namespace std;

template <class T>
StackType<T>::StackType()
{
    head = NULL;
}

template <class T>
bool StackType<T>::IsEmpty()
{
    return (head == NULL);
}
```

```
template <class T>
bool StackType<T>::IsFull()
{
    try
    {
        Node* temp = new Node;
        delete temp;
        return false;
    }
    catch (bad_alloc &exception)
    {
        return true;
    }
}

template <class T>
void StackType<T>::Push(T value)
{
    if (IsFull())
        throw FullStack();
    else
    {

        Node* temp = new Node;
        temp->data = value;
        temp->next = head;
        head = temp;
    }
}

template <class T>
void StackType<T>::Pop()
{
    if (IsEmpty())
        throw EmptyStack();
    else
    {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}

template <class T>
T StackType<T>::Top()
{
    if (IsEmpty())
        throw EmptyStack();
    else
        return head->data;
}

template <class T>
StackType<T>::~StackType()
{
    Node* i = head;
    Node* nextNode;

    while (i != NULL)
    {
        nextNode = i->next; // Store the next node
        delete i;           // Delete the current node
        i = nextNode;       // Move to the next node
    }
}
```

```
template <class T>
void StackType<T>::Diagnose()
{
    Node* i = head;
    while (i != NULL)
    {
        cout << "self: " << i << ", data: " << i->data << ", next: " << i->next << endl;
        i = i->next;
    }
}
```

STACK (LINKED LIST–BASED)

TASKS:

Instructions:

- Create the driver file (main.cpp) and perform the following tasks.
- You cannot make any changes to the header (**.h**) or source (**.cpp**) files of the **StackType** class.

OPERATION	INPUT VALUES	EXPECTED OUTPUT
Create a stack of integers		
Check if the stack is empty		Stack is Empty
Push four items	5, 7, 4, 2	
Check if the stack is empty		Stack is not Empty
~~Check if the stack is full~~		~~Stack is not full~~
Print the values in the stack (in the order the values are given)		5, 7, 4, 2
Push another item	3	
Print the values in the stack		5, 7, 4, 2, 3
~~Check if the stack is full~~		~~Stack is full~~
Pop two items		
Print top item		4

OPERATION	INPUT VALUES	EXPECTED OUTPUT
Task: Take an infix expression as input from the user, determine the outcome of the expression, and return the result as output. If the expression is invalid, return the text "Invalid expression.". You will solve this problem in two steps: ○ **Convert the infix expression to postfix notation.** You will need a stack to perform this conversion. ○ **Evaluate the postfix expression to determine the final result.** Again, you will need a stack to evaluate the expression. The operands in the infix expression are single-digit non-negative numbers, and the operators include addition (+), subtraction (-), multiplication (*), and division (/).	`10 + 3 * 5 / (16 - 4)`	`10 3 5 * 16 4 - / +` `11.25`
	`(5 + 3) * 12 / 3`	`5 3 + 12 * 3 /` `32`
	`3 + 4 / (2 - 3) * / 5`	**Invalid Expression**
	`7 / 5 + (4 - (2) * 3`	**Invalid Expression**