CSE225L – Data Structures and Algorithms Lab
Lab 10
Stack (Linked-list based)

In today's lab we will design and implement the Stack ADT using linked-list.

stacktype.h

```
#ifndef STACKTYPE_H
#define STACKTYPE_H

class FullStack
{};

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;
    }
}
```

Generate the **driver file (main.cpp)** where you perform the following tasks. Note that you cannot make any change to the header file or the source file.

(Optional Task)		
Operation to Be Tested and Description of Action	**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
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 not full
Pop two items		
Print top item		4

Take strings of parentheses as input from the user and use a stack to check if each string is balanced.	( )	Balanced
	(( )) ( ) (( ) ( )) ( )	Balanced
	(( )) ( ) ((( )	Not Balanced
	(( )))) ((( )	Not Balanced
	(( ))))))))	Not Balanced

(Main Task)		
Operation to Be Tested and Description of Action	**Input Values**	**Expected Output**

Take infix expressions from the user as input, determine the outcome of the expression and gives that back to user as output, or the text "Invalid expression" if the expression is not a valid one. You will have to solve this problem in two steps.
▪ First, you have to convert the expression from infix notation to postfix notation. You are going to need a stack in order to do so.
▪ In the next step, you will have to evaluate the postfix expression and determine the final result. Again, you will need a stack in order to do this.
All the operands in the infix expressions are single digit non-negative operands 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`**