Міністерство освіти і науки України
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені ІВАНА ФРАНКА
Факультет прикладної математики та інформатики

Кафедра програмування

ЛАБОРАТОРНА РОБОТА № 2

Стек та Стек-STL

з курсу “Алгоритми та структури даних”

Виконав:

Студент групи ПМІ-16

Бевз Маркіян Юрійович

Львів - 2024

Мета: Дослідити роботу стеку як однієї з ключових структур даних. Зрозуміти принцип роботи стеку, що розташований у стандартній бібліотеці та реалізувати свій власний стек(реалізовуватиму варіант, що базуватиметься на однозв’язному списку).

Стек - це структура даних, яка працює за принципом "останній увійшов - перший вийшов" (Last In, First Out, LIFO). Це означає, що останній елемент, який додається до стеку, буде першим, що вийде зі стеку при видаленні елементів. Операції над стеком обмежуються додаванням елементу на вершину стеку (**Push**) та видаленням верхнього елемента (**Pop**).

Однозв'язний список - це структура даних, де кожен елемент (вузол) має посилання на наступний елемент у списку. У контексті стеку на основі однозв'язного списку, вершиною стеку буде початок (голова) однозв'язного списку. Такий стек буде представлений як послідовність вузлів, де кожен вузол має посилання на свого наступника.

Принцип роботи стека на основі однозв'язного списку можна розглядати так:

1. **Додавання елементу (Push):**
    - Створюється новий вузол з необхідним значенням.
    - Посилання на наступника цього нового вузла встановлюється на поточну вершину стеку.
    - Новий вузол стає новою вершиною стеку.
2. **Видалення елементу (Pop):**
    - Перевіряється, чи стек не порожній.
    - Якщо стек порожній, можна повернути помилку або просто нічого не робити (залежно від реалізації).
    - Якщо стек не порожній, вершина стеку видаляється.
    - Нова вершина стеку стає тим вузлом, на який посилалася попередня вершина.
3. **Отримання верхнього елементу (Top):**
    - Перевіряється, чи стек не порожній.
    - Якщо стек порожній, можна повернути помилку або просто нічого не робити.
    - Якщо стек не порожній, повертається значення верхнього елементу (значення вершини стеку).
4. **Очищення стеку (Clear):**

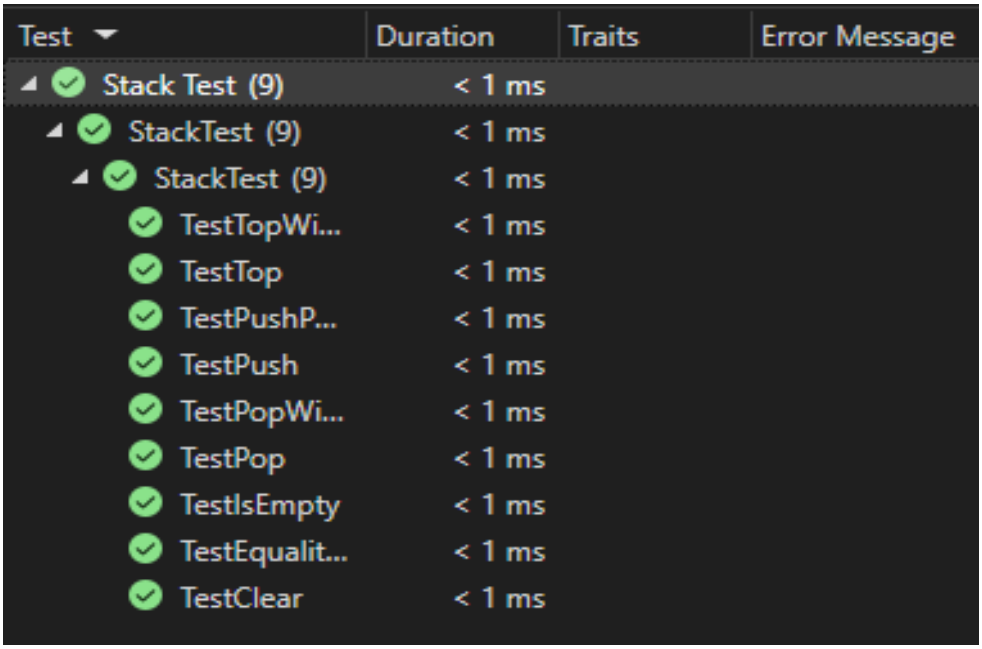
- Поки стек не порожній, видаляється вершина стеку (використовуючи операцію **Pop**).

Такий підхід дозволяє ефективно використовувати однозв'язний список для реалізації стеку та виконувати основні операції стеку у часі O(1), саме тому я його й вибрав.

Після того, як ретельно дослідив принцип роботи стеку зі стандартної бібліотеки я реалізував свій власний стек на основі однозв’язного списку, як і вказував вище. Ось зображення результату роботи моєї програми, можна зауважити, що працює так само, як і стек стандартної бібліотеки.

```
How many numbers do you want in your stack?: 6
12 23 34 45 56 67
Is STL stack empty? [false]
67
Is My stack empty? [false]
67
Is STL stack empty? [false]
56
Is My stack empty? [false]
56
Is STL stack empty? [false]
45
Is My stack empty? [false]
45
Is STL stack empty? [false]
34
Is My stack empty? [false]
34
Is STL stack empty? [false]
23
Is My stack empty? [false]
23
Is STL stack empty? [false]
12
Is My stack empty? [false]
12
Is STL stack empty? [true]Is My stack empty? [true]
```

Також реалізував тести, що перевіряють роботу усіх основних методів стеку у різних ситуаціях, загалом 9 тестів. Нижче подано зображення результату виконання та самих тестів.




```cpp
TEST_CLASS(StackTest)
{
public:
    TEST_METHOD(TestClear)
    {
        Stack TestStack;
        for (int i = 1; i <= 10; ++i) {
            TestStack.Push(i * 2);
        }
        Assert::IsFalse(TestStack.isEmpty());
        TestStack.Clear();
        Assert::IsTrue(TestStack.isEmpty());
    }
    TEST_METHOD(TestPush)
    {
        Stack TestStack;
        TestStack.Push(10);

        Assert::AreEqual(TestStack.Top(), 10);
        Assert::IsFalse(TestStack.isEmpty());
    }
    TEST_METHOD(TestPop)
    {
        Stack TestStack;
        TestStack.Push(20);
        TestStack.Push(30);

        TestStack.Pop();
        Assert::AreEqual(TestStack.Top(), 20);

        TestStack.Pop();
        Assert::IsTrue(TestStack.isEmpty());
    }
```

```
TEST_METHOD(TestTop)
{
    Stack TestStack;
    Assert::IsTrue(TestStack.isEmpty());

    TestStack.Push(42);
    Assert::AreEqual(TestStack.Top(), 42);

    TestStack.Push(55);
    Assert::AreEqual(TestStack.Top(), 55);
}

TEST_METHOD(TestIsEmpty)
{
    Stack TestStack;
    Assert::IsTrue(TestStack.isEmpty());

    TestStack.Push(5);
    Assert::IsFalse(TestStack.isEmpty());

    TestStack.Pop();
    Assert::IsTrue(TestStack.isEmpty());
}

TEST_METHOD(TestPushPopWithMultipleValues)
{
    Stack TestStack;

    for (int i = 1; i <= 100; ++i) {
        TestStack.Push(i);
    }

    Assert::AreEqual(TestStack.Top(), 100);
    Assert::IsFalse(TestStack.isEmpty());

    TestStack.Clear();

    Assert::IsTrue(TestStack.isEmpty());
}

TEST_METHOD(TestTopWithEmptyStack)
{
    Stack TestStack;

    Assert::AreEqual(TestStack.Top(), 0);
    Assert::IsTrue(TestStack.isEmpty());
}
```

```
TEST_METHOD(TestEqualityWithSTLStack)
{
    Stack TestStack;
    for (int i = 1; i <= 50; ++i) {
        TestStack.Push(i);
    }
    std::stack<int>  STLStack;
    for (int i = 1; i <= 50; ++i) {
        STLStack.push(i);
    }
    TestStack.Clear();
    STLStack = {};

    Assert::IsTrue(TestStack.isEmpty());
    Assert::IsTrue(STLStack.empty());
}


TEST_METHOD(TestPopWithEmptyStack)
{
    Stack TestStack;

    TestStack.Pop();

    Assert::IsTrue(TestStack.isEmpty());
}
```

Висновок: На цій лабораторній роботі я дослідив роботу стеку як однієї з ключових структур даних, що використовується при реалізації багатьох алгоритмів(до прикладу алгоритми обходу графів та дерева). Зрозумів принцип роботи стеку, що розташований у стандартній бібліотеці та реалізував свій власний стек, що базується на однозв’язному списку.