МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ



**Дніпровський національний університет  
залізничного транспорту імені академіка В. Лазаряна**

Кафедра «Комп'ютерні інформаційні технології»

**Лабораторна робота №11**

**з дисципліни «Основи програмування»**

**на тему: «**Лінійні динамічні структури**»**

Виконав:

студент гр.ПЗ1911

Сафонов Д.Є.

Прийняла:

Нежуміра О. І.

Дніпро, 2020

**Тема.** Лінійні динамічні структури.

**Мета.** Навчитися реалізовувати лінійні динамічні структури. Оволодіти навичками використання лінійних динамічних структур в розробці програм.

# **Постановка завдання (завдання до лабораторної роботи, індивідуальне завдання).**

Розробити програму для заданого індивідуального завдання і обраного рівня складності.

Вимоги до програми:

− валідація вхідних даних;

− програма повинна мати багатомодульну структуру, яка базується на реалізації функціонала програми.

Вимоги до тексту програми:

− специфікації програми і її функцій;

− самодокументованість коду: всі ідентифікатори повинні мати назви, що відповідають суті змінних.

Рівень II (82-100 балів): в завданні вказана лише предметна область, кількість балів, якими буде оцінено завдання, залежить від правильно обраних структур даних, об’єму реалізованого в рамках завдання функціоналу та його якості. Використання файлів для зберігання даних, ведення протоколів роботи програми тощо буде враховано.

7. Калькулятор «великих» цілих чисел.

# **2. Зовнішні специфікації програми (вхідні та вихідні дані, функціональні вимоги до програми).**

2.1. Вхідні дані:

|  |  |  |  |
| --- | --- | --- | --- |
| Найменування даних | Умовне позначення | Вимоги до данних | Приклад |
| Вхідний рядок | infix | Математичний вираз.  Токени розділені одним пробілом.  Початок рядку – токен.  Кінець рядку – токен.  Токенами можуть бути операнди та оператори.  Операнд – ціле число у десятковій системі з опціональним унарним + або – на початку, не може починатися з нуля якщо це не единий символ.  Оператори: +, -, \*, /. | 2 \* 3 – 4 \* 5 + 6 /3 |
| Вибір виход чи продовження | exit\_ | Один з символів: Y, N, y, n | y |

2.2. Вихідні дані:

|  |  |  |  |
| --- | --- | --- | --- |
| Найменування даних | Умовне позначення | Вимоги до данних | Приклад |
| результат | infix(приклад та результат зберігаються у одному об’єкті почергово) | Рядок, ціле число або “NaN”,або повідомлення про помилку | -100 |

2.3. Функціональні вимоги:

- Отримання вхідного рядка;

- Валідація вхідного рядка;

- Розрахунок результату;

- Повернення розвернутих повідомлень про помилку у вхідному рядку.

# **3. Опис структур даних, використаних для реалізації завдання.**

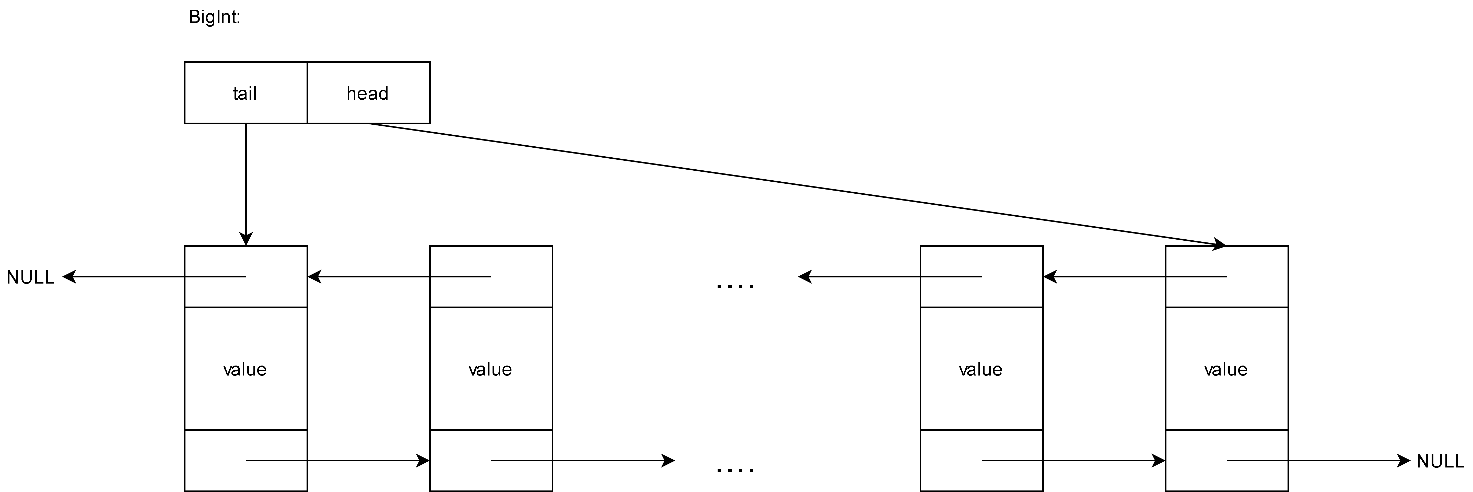
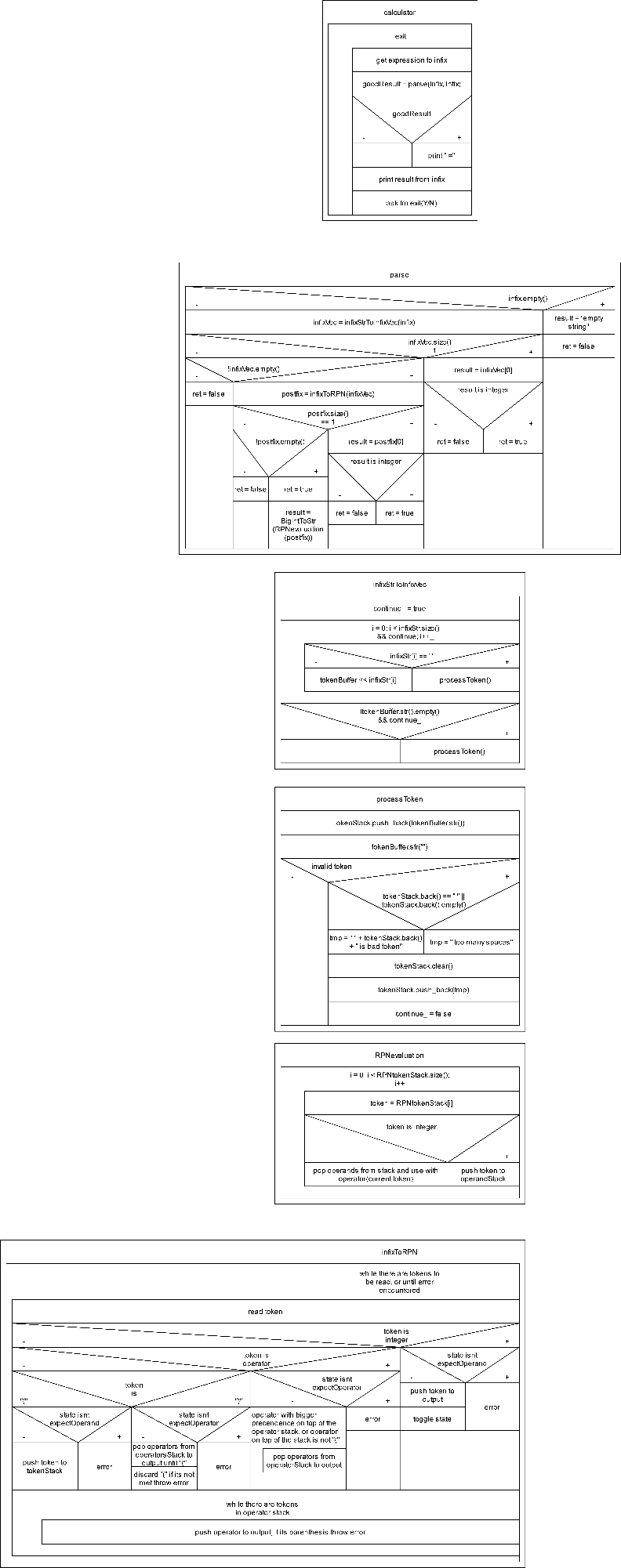
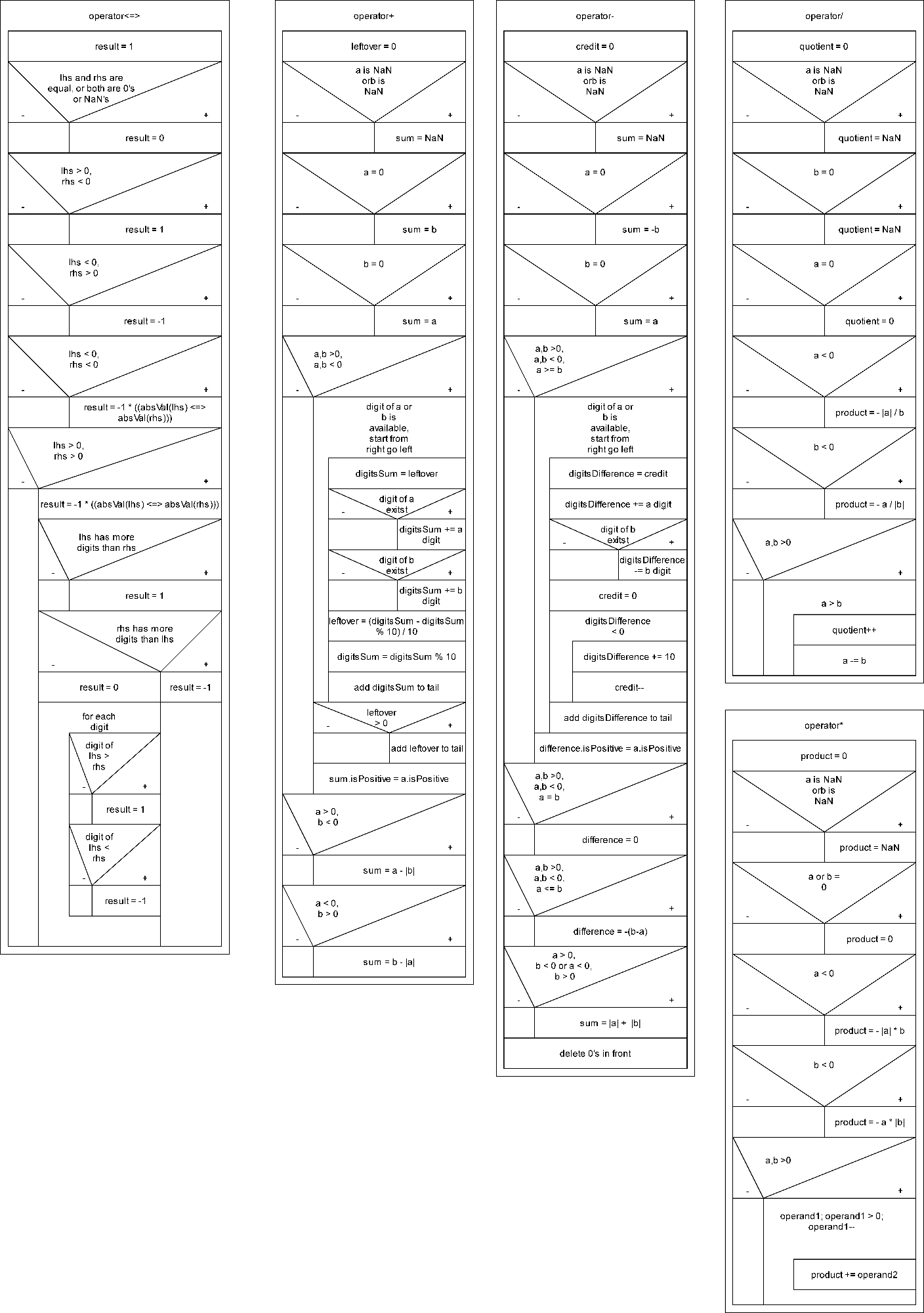


Рисунок 1

# **4. Алгоритм розв’язання завдання (алгоритм треба розбити на частини за функціоналом і представити у вигляді діаграм Нассі-Шнейдермана).**



# **5. Набори тестів для перевірки правильності виконання програми.**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Назва | Вхідні дані | Очікуваний результат |
| 1 | Додавання | 2 + 3 | 5 |
| 2 | Множення | 2 \* 3 | 6 |
| 3 | Один операнд | 89 | 89 |
| 4 | Порожній рядок |  | Повідомлення про помилку |
| 5 | Забагато пробілів | 12 - 4 | Повідомлення про помилку |
| 6 | Відсутні пробіли | 12-4 | Повідомлення про помилку |
| 7 | Складний приклад | 2 \* 3 - 4 \* 5 + 6 / 3 | -12 |
| 8 | Некоректний токен | 6 + с або 1- abc | Повідомлення про помилку |
| 9 | Порушення синтаксису | 5 + + 6 | Повідомлення про помилку |
| 10 | Ділення на нуль | 5 / 0 | NaN |
| 11 | Один операнд у скобках | ( 2 ) | 2 |
| 12 | Складний приклад2 | 5 + 2 \* 3 - 1 + 7 \* 8 | 66 |
| 13 | Один операнд у багатьох скобках | ( ( ( 4 ) ) ) | 4 |
| 14 | Не закриті скобки | ( ( 4 ) | Повідомлення про помилку |
| 15 | Не відкриті скобки | ( 4 ) ) | Повідомлення про помилку |

# **6. Текст програми.**

“main.cpp”

//This app is basic arbitrary precision integer only calculator

//input is infix expression with tokens separated by single whitespace

//tokens are:

// operators:

// + addition

// - substraction

// \* multiplication

// / division

// unary + - are allowed but only for constants(eg -101, +0, etc)

// operands:

// optional +-, can start from 0 if its the only digit, base10

#include "infixParser.h"

#include "tests.h"

int main() {

//operatorPlusTest();

//operatorMinusTest();

//operatorMultiplyTest();

//operatorDivisionTest();

//parserTest();

calculator();

return 0;

}

“tests.h”

#ifndef \_\_TESTS\_H\_\_

#define \_\_TESTS\_H\_\_

//tests for operators of BigInt class

//tests for parser that uses BigInt class

void operatorPlusTest();

void operatorMinusTest();

void operatorMultiplyTest();

void operatorDivisionTest();

void parserTest();

#endif

“tests.cpp”

#include "tests.h"

#include "BigInt.h"

#include "infixParser.h"

#include <iostream>

void operatorPlusTest() {

std::cout << "OPERATOR+" << std::endl;

BigInt arbitraryNumber1, arbitraryNumber2;

std::cout << "+ + |a|>|b| :\t";

arbitraryNumber1 = "6945292007472";

arbitraryNumber2 = "987654321";

std::cout << BigIntToStr(arbitraryNumber1) << "\t + \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "+ + |a|<|b| :\t";

arbitraryNumber1 = "987654321";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t + \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "+ - |a|>|b| :\t";

arbitraryNumber1 = "6945292007472";

arbitraryNumber2 = "-987654321";

std::cout << BigIntToStr(arbitraryNumber1) << "\t + \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "+ - |a|<|b| :\t";

arbitraryNumber1 = "987654321";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t + \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "+ +0 :\t\t";

arbitraryNumber1 = "987654321";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t + \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "+ -0 :\t\t";

arbitraryNumber1 = "987654321";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t + \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

//

std::cout << "- + |a|>|b| :\t";

arbitraryNumber1 = "-6945292007472";

arbitraryNumber2 = "987654321";

std::cout << BigIntToStr(arbitraryNumber1) << "\t + \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "- + |a|<|b| :\t";

arbitraryNumber1 = "-987654321";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t + \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "- - |a|>|b| :\t";

arbitraryNumber1 = "-6945292007472";

arbitraryNumber2 = "-987654321";

std::cout << BigIntToStr(arbitraryNumber1) << "\t + \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "- - |a|<|b| :\t";

arbitraryNumber1 = "-987654321";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t + \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "- +0 :\t\t";

arbitraryNumber1 = "-987654321";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t + \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "- -0 :\t\t";

arbitraryNumber1 = "-987654321";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t + \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

//

std::cout << "+0 + :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t + \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "+0 - :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t + \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "+0 +0 :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t + \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "+0 -0 :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t + \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

//

std::cout << "-0 + :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t + \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "-0 - :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t + \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "-0 +0 :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t + \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "-0 -0 :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t + \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

std::cout << "single NaN test:";

BigInt NaN;

arbitraryNumber1 = NaN;

arbitraryNumber2 = "987";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t + \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 + arbitraryNumber2) << std::endl;

}

void operatorMinusTest() {

std::cout << "OPERATOR-" << std::endl;

BigInt arbitraryNumber1, arbitraryNumber2;

std::cout << "+ + |a|>|b| :\t";

arbitraryNumber1 = "6945292007472";

arbitraryNumber2 = "987654321";

std::cout << BigIntToStr(arbitraryNumber1) << "\t - \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "+ + |a|<|b| :\t";

arbitraryNumber1 = "987654321";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t - \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "+ - |a|>|b| :\t";

arbitraryNumber1 = "6945292007472";

arbitraryNumber2 = "-987654321";

std::cout << BigIntToStr(arbitraryNumber1) << "\t - \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "+ - |a|<|b| :\t";

arbitraryNumber1 = "987654321";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t - \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "+ +0 :\t\t";

arbitraryNumber1 = "987654321";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t - \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "+ -0 :\t\t";

arbitraryNumber1 = "987654321";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t - \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

//

std::cout << "- + |a|>|b| :\t";

arbitraryNumber1 = "-6945292007472";

arbitraryNumber2 = "987654321";

std::cout << BigIntToStr(arbitraryNumber1) << "\t - \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "- + |a|<|b| :\t";

arbitraryNumber1 = "-987654321";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t - \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "- - |a|>|b| :\t";

arbitraryNumber1 = "-6945292007472";

arbitraryNumber2 = "-987654321";

std::cout << BigIntToStr(arbitraryNumber1) << "\t - \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "- - |a|<|b| :\t";

arbitraryNumber1 = "-987654321";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t - \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "- +0 :\t\t";

arbitraryNumber1 = "-987654321";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t - \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "- -0 :\t\t";

arbitraryNumber1 = "-987654321";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t - \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

//

std::cout << "+0 + :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t - \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "+0 - :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t - \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "+0 +0 :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t - \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "+0 -0 :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t - \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

//

std::cout << "-0 + :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t - \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "-0 - :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t - \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "-0 +0 :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t - \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "-0 -0 :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t - \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "single NaN test:";

BigInt NaN;

arbitraryNumber1 = NaN;

arbitraryNumber2 = "987";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t - \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

std::cout << "a = b:\t\t";

arbitraryNumber1 = arbitraryNumber2;

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t - \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 - arbitraryNumber2) << std::endl;

}

void operatorMultiplyTest() {

std::cout << "OPERATOR\*" << std::endl;

BigInt arbitraryNumber1, arbitraryNumber2;

std::cout << "+ + |a|>|b| :\t";

arbitraryNumber1 = "6945292007472";

arbitraryNumber2 = "987";

std::cout << BigIntToStr(arbitraryNumber1) << "\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "+ + |a|<|b| :\t";

arbitraryNumber1 = "987";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "+ - |a|>|b| :\t";

arbitraryNumber1 = "6945292007472";

arbitraryNumber2 = "-987";

std::cout << BigIntToStr(arbitraryNumber1) << "\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "+ - |a|<|b| :\t";

arbitraryNumber1 = "987";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "+ +0 :\t\t";

arbitraryNumber1 = "987";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "+ -0 :\t\t";

arbitraryNumber1 = "987";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

//

std::cout << "- + |a|>|b| :\t";

arbitraryNumber1 = "-6945292007472";

arbitraryNumber2 = "987";

std::cout << BigIntToStr(arbitraryNumber1) << "\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "- + |a|<|b| :\t";

arbitraryNumber1 = "-987";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "- - |a|>|b| :\t";

arbitraryNumber1 = "-6945292007472";

arbitraryNumber2 = "-987";

std::cout << BigIntToStr(arbitraryNumber1) << "\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "- - |a|<|b| :\t";

arbitraryNumber1 = "-987";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "- +0 :\t\t";

arbitraryNumber1 = "-987";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "- -0 :\t\t";

arbitraryNumber1 = "-987";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

//

std::cout << "+0 + :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "+0 - :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "+0 +0 :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "+0 -0 :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

//

std::cout << "-0 + :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "-0 - :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "-0 +0 :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "-0 -0 :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

std::cout << "single NaN test:";

BigInt NaN;

arbitraryNumber1 = NaN;

arbitraryNumber2 = "987";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t \* \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 \* arbitraryNumber2) << std::endl;

}

void operatorDivisionTest() {

std::cout << "OPERATOR/" << std::endl;

BigInt arbitraryNumber1, arbitraryNumber2;

std::cout << "+ + |a|>|b| :\t";

arbitraryNumber1 = "6945292007472";

arbitraryNumber2 = "694529200";

std::cout << BigIntToStr(arbitraryNumber1) << "\t / \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "+ + |a|<|b| :\t";

arbitraryNumber1 = "694529200";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t / \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "+ - |a|>|b| :\t";

arbitraryNumber1 = "6945292007472";

arbitraryNumber2 = "-694529200";

std::cout << BigIntToStr(arbitraryNumber1) << "\t / \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "+ - |a|<|b| :\t";

arbitraryNumber1 = "694529200";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t / \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "+ +0 :\t\t";

arbitraryNumber1 = "694529200";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t / \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "+ -0 :\t\t";

arbitraryNumber1 = "694529200";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t / \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

//

std::cout << "- + |a|>|b| :\t";

arbitraryNumber1 = "-6945292007472";

arbitraryNumber2 = "694529200";

std::cout << BigIntToStr(arbitraryNumber1) << "\t / \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "- + |a|<|b| :\t";

arbitraryNumber1 = "-694529200";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t / \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "- - |a|>|b| :\t";

arbitraryNumber1 = "-6945292007472";

arbitraryNumber2 = "-694529200";

std::cout << BigIntToStr(arbitraryNumber1) << "\t / \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "- - |a|<|b| :\t";

arbitraryNumber1 = "-694529200";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t / \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "- +0 :\t\t";

arbitraryNumber1 = "-694529200";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t / \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "- -0 :\t\t";

arbitraryNumber1 = "-694529200";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t / \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

//

std::cout << "+0 + :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t / \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "+0 - :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t / \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "+0 +0 :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t / \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "+0 -0 :\t\t";

arbitraryNumber1 = "0";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t / \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

//

std::cout << "-0 + :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t / \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "-0 - :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "-6945292007472";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t / \t" << BigIntToStr(arbitraryNumber2) << "\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "-0 +0 :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t / \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "-0 -0 :\t\t";

arbitraryNumber1 = "-0";

arbitraryNumber2 = "-0";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t / \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

std::cout << "single NaN test:";

BigInt NaN;

arbitraryNumber1 = NaN;

arbitraryNumber2 = "987";

std::cout << BigIntToStr(arbitraryNumber1) << "\t\t / \t" << BigIntToStr(arbitraryNumber2) << "\t\t = \t" << BigIntToStr(arbitraryNumber1 / arbitraryNumber2) << std::endl;

}

void parserTest() {

std::vector<std::string> tests{

"2 + 3",

"2 \* 3",

"89",

"",

"12 - 4",

"12-4",

"2 \* 3 - 4 \* 5 + 6 / 3",

"6 + c",

"1 - abc",

"5 + + 6",

"5 / 0",

"( 2 )",

"5 + 2 \* 3 - 1 + 7 \* 8",

"( ( ( 4 ) ) )",

"( ( 4 )",

"( 4 ) )"

};

for (auto expression : tests) {

std::string result;

bool goodResult = parse(expression, result);

std::cout << expression;

if (goodResult)

std::cout << " = " << result << std::endl;

else

std::cout << result << std::endl;

}

}

“infixParser.h”

#ifndef \_\_INFIXPARSER\_H\_\_

#define \_\_INFIXPARSER\_H\_\_

#include <vector>

#include "BigInt.h"

//tokens separated by whitespaces//coverts infix token vector to postfix

std::vector<std::string> infixToRPN(const std::vector<std::string>& infix);

//evaluates postfix token vector

BigInt RPNevaluation(const std::vector<std::string>& RPNtokenStack);

//converts infix string to vector of tokens in order

std::vector<std::string> infixStrToInfixVec(std::string infixStr);

//returns true if input is good, else - false,result contains result of evaluation or commentary about bad input

bool parse(const std::string& infix, std::string& result);

void calculator();

#endif

“infixParser.cpp”

#include "infixParser.h"

#include <vector>

#include <sstream>

#include <iostream>

#include <cmath>

#include <algorithm>

#include <iterator>

#include <cstdlib>

#include <map>

#include <regex>

#include "BigInt.h"

//tokens separated by whitespace

std::vector<std::string> infixToRPN(const std::vector<std::string>& infix) {

std::map<std::string, int> operatorPrecendence{

{"+", 2},

{"-", 2},

{"/", 3},

{"\*", 3}

};

std::vector<std::string> operatorStack;

std::string token;

std::vector<std::string> RPN;

std::regex integer("^([+-]?[0-9]|([1-9][0-9]\*))$");

enum state { expectOperator, expectOperand }currentState = expectOperand;

bool continue\_ = true;

for (int i = 0; i < infix.size() && continue\_; i++) {//while there are tokens to be read

token = infix[i];//read a token

if (std::regex\_match(token, integer)) {//token is integer

if (currentState != expectOperand) {

RPN.clear();

RPN.push\_back(" syntax error");

continue\_ = false;//return error

}

else {

RPN.push\_back(token);

currentState = expectOperator;

}

}

else if (operatorPrecendence.count(token)) {//token is operator

if (currentState != expectOperator) {

RPN.clear();

RPN.push\_back(" syntax error");

continue\_ = false;//return error

}

else {

//operator with bigger precendence on top of the operator stack, or operator on top of the stack is not "("

while ((operatorStack.size() > 0) && operatorStack.back() != "(" && (operatorPrecendence.at(token) <= operatorPrecendence.at(operatorStack.back()))) {

RPN.push\_back(operatorStack.back()); operatorStack.pop\_back();

}

operatorStack.push\_back(token);

currentState = expectOperand;

}

}

else if (token == "(")

if (currentState != expectOperand) {

RPN.clear();

RPN.push\_back(" syntax error");

continue\_ = false;//return error

}

else

operatorStack.push\_back(token);//state remains the same

else if (token == ")") {

if (currentState != expectOperator) {

RPN.clear();

RPN.push\_back(" syntax error");

continue\_ = false;//return error

}

else {

while (operatorStack.size() > 0 && operatorStack.back() != "(") {//check if can find "("

RPN.push\_back(operatorStack.back()); operatorStack.pop\_back();

}

if (operatorStack.size() > 0)

operatorStack.pop\_back();//discarding "("

else {

RPN.clear();

RPN.push\_back("error, mismatched parenthesis");

continue\_ = false;//return error

}

//state remains the same

}

}

}

while (operatorStack.size() > 0 && continue\_) {

if (operatorStack.back() == "(" || operatorStack.back() == ")") {

RPN.clear();

RPN.push\_back("error, mismatched parenthesis");

continue\_ = false;//return error

}

else

RPN.push\_back(operatorStack.back()); operatorStack.pop\_back();

}

return RPN;

}

BigInt RPNevaluation(const std::vector<std::string>& RPNtokenStack) {//make pre validation

std::vector<BigInt> operandStack;

std::string token;

std::regex integer("^([+-]?[0-9]|([1-9][0-9]\*))$");

for (int i = 0; i < RPNtokenStack.size(); i++) {//while tokens available

token = RPNtokenStack[i];

if (std::regex\_match(token, integer))//token is integer

operandStack.push\_back(token);

else {//token is operator

BigInt operand1 = operandStack.back(); operandStack.pop\_back();

BigInt operand2 = operandStack.back(); operandStack.pop\_back();

if (token == "\*")

operandStack.push\_back(operand2 \* operand1);

else if (token == "/")

operandStack.push\_back(operand2 / operand1);

else if (token == "-")

operandStack.push\_back(operand2 - operand1);

else if (token == "+")

operandStack.push\_back(operand2 + operand1);

}

}

return operandStack.back();

}

std::vector<std::string> infixStrToInfixVec(std::string infixStr) {

std::map<std::string, int> operatorPrecendence{

{"+", 2},

{"-", 2},

{"/", 3},

{"\*", 3},

{"(", 0},

{")", 0}

};

std::stringstream tokenBuffer;

std::vector<std::string> tokenStack;

bool continue\_ = true;

std::regex integer("^([+-]?[0-9]|([1-9][0-9]\*))$");

auto processToken = [&]() {

tokenStack.push\_back(tokenBuffer.str());

tokenBuffer.str("");

if (!(std::regex\_match(tokenStack.back(), integer) || operatorPrecendence.count(tokenStack.back()))) {//if token isnt operator or integer

std::string tmp;

if (tokenStack.back() == " " || tokenStack.back().empty())

tmp = " too many spaces";

else

tmp = " " + tokenStack.back() + " is bad token";//to know which one is bad

tokenStack.clear();

tokenStack.push\_back(tmp);

continue\_ = false;

}

};//to make code smaller

for (int i = 0; i < infixStr.size() && continue\_; i++)

if (infixStr[i] == ' ')

processToken();

else

tokenBuffer << infixStr[i];

if (!tokenBuffer.str().empty() && continue\_)

processToken();

return tokenStack;

}

bool parse(const std::string& infix, std::string& result) {//true if good input

std::regex integer("^([+-]?[0-9]|([1-9][0-9]\*))$");

bool ret;

if (infix.empty()) {

result = "empty string";

ret = false;

}

else {

std::vector<std::string> infixVec;

infixVec = infixStrToInfixVec(infix);//infix string to infix vector of tokens

if (infixVec.size() == 1) {

result = infixVec[0];

if (std::regex\_match(infixVec[0], integer))

ret = true;

else

ret = false;

}

else if (!infixVec.empty()) {

std::vector<std::string> postfix = infixToRPN(infixVec);//infix vector of tokens to postfix vetor of tokens

if (postfix.size() == 1) {

result = postfix[0];

if (std::regex\_match(postfix[0], integer))

ret = true;

else

ret = false;

}

else if (!postfix.empty()) {

result = BigIntToStr(RPNevaluation(postfix));//evaluate postfix

ret = true;

}

else

ret = false;

}

else

ret = false;

}

return ret;

}

void calculator() {

bool exit\_ = false;

while (!exit\_) {

std::string infix;

std::cout << "input math expression:" << std::endl;

std::getline(std::cin, infix);

bool goodResult = parse(infix, infix);

if (goodResult)

std::cout << " =";

std::cout << " " << infix << std::endl;

char chExit;

std::cout << "Do you want to exit the programm?(Y/N): ";

std::cin >> chExit;

while ((chExit != 'Y' && chExit != 'y' && chExit != 'N' && chExit != 'n') || (std::cin.peek() != '\n')) {//until correct input

while (std::cin.get() != '\n');//clear istream

std::cout << "wrong input, try again." << std::endl;

std::cout << "Do you want to exit the programm?(Y/N): ";

std::cin >> chExit;

}

if (chExit == 'Y' || chExit == 'y')

exit\_ = true;

else

exit\_ = false;

std::cin.get();//enter

}

}

“BigInt.h”

#ifndef \_\_BIGINT\_H\_\_

#define \_\_BIGINT\_H\_\_

//class that represents arbitrary precision integer using two way linked list

//no digits represent NaN

//+-0 is still 0

#include <string>

#include <compare>

struct digit {

char value;//0-9

digit\* next, \* prev;

};

class BigInt {//tail <-> digits <-> head

public:

BigInt();

~BigInt();

BigInt(const std::string& rhs);

BigInt(const BigInt& rhs);

BigInt& operator=(const std::string& rhs);

BigInt& operator=(const BigInt& rhs);

BigInt operator+() const;//pass

BigInt operator-() const;//reverse isPositive

friend std::string BigIntToStr(const BigInt& toString);

friend int operator<=>(const BigInt& lhs, const BigInt& rhs);

friend BigInt operator+(const BigInt& a, const BigInt& b);

friend BigInt operator-(const BigInt& a, const BigInt& b);

friend BigInt operator\*(const BigInt& a, const BigInt& b);

friend BigInt operator/(const BigInt& a, const BigInt& b);

friend BigInt absVal(const BigInt& a);

private:

digit\* head, \* tail;//if head = tail = NULL then NaN

bool isPositive;//0 can be both negative and positive

void addToTail(const char& value);

void deleteTail();

void deleteDigits();

};

std::string BigIntToStr(const BigInt& toString);//return string representation in base10////sign unnecessary <-> tail <-> digits <-> head

int operator<=>(const BigInt& lhs, const BigInt& rhs);//three-way comparison

BigInt operator+(const BigInt& a, const BigInt& b);

BigInt operator-(const BigInt& a, const BigInt& b);

BigInt operator\*(const BigInt& a, const BigInt& b);

BigInt operator/(const BigInt& a, const BigInt& b);

BigInt absVal(const BigInt& a);

#endif

“BigInt.cpp”

#include "BigInt.h"

#include <sstream>//stringstream

BigInt::BigInt() {

this->head = this->tail = NULL;

}

int operator<=>(const BigInt& lhs, const BigInt& rhs) {

BigInt zero;

zero = "0";

int result = 1;

if ((BigIntToStr(lhs) == BigIntToStr(rhs)) || ((BigIntToStr(absVal(lhs)) == "0") && (BigIntToStr(absVal(rhs)) == "0")) || (BigIntToStr(lhs) == "NaN") || (BigIntToStr(rhs) == "NaN"))//a=b, a=+-0, b=+-0 or one of them is NaN

result = 0;

else if (lhs.isPositive && !rhs.isPositive)//a>0, b<0, a>b

result = 1;

else if (!lhs.isPositive && rhs.isPositive)//a<0, b>0, a<b

result = -1;

else if (!lhs.isPositive && !rhs.isPositive)//a,b<0

result = -1 \* (absVal(lhs) <=> absVal(rhs));

else {//a,b>0

if (BigIntToStr(lhs).length() > BigIntToStr(rhs).length())//a>b

result = 1;

else if (BigIntToStr(lhs).length() < BigIntToStr(rhs).length())//a<b

result = -1;

else {//same length

result = 0;

std::string lhsStr = BigIntToStr(lhs);

std::string rhsStr = BigIntToStr(rhs);

int digitsAmount = lhsStr.length();

for (int i = 0; i < digitsAmount && result == 0; i++) {

if (lhsStr[i] > rhsStr[i])

result = 1;

if (lhsStr[i] < rhsStr[i])

result = -1;

}

}

}

return result;

}

BigInt& BigInt::operator= (const std::string& rhs) {

this->deleteDigits();

for (int i = rhs.size() - 1; i > 0; i--)

this->addToTail(rhs[i] - '0');

if (rhs[0] == '+')

this->isPositive = true;

else if (rhs[0] == '-')

this->isPositive = false;

else {//char is digit//actually any other char

this->isPositive = true;

addToTail(rhs[0] - '0');

}

return \*this;

}

std::string BigIntToStr(const BigInt& toString) {

std::string string\_;

std::stringstream sstring\_;

if (!toString.isPositive)

sstring\_ << '-';

digit\* currentDigit = toString.tail;

if (currentDigit) {

while (currentDigit) {

sstring\_ << static\_cast<char>(currentDigit->value + '0');

currentDigit = currentDigit->prev;

}

string\_ = sstring\_.str();

if (string\_ == "-0")

string\_ = "0";

}

else

string\_ = "NaN";

return string\_;

}

void BigInt::addToTail(const char& value) {

digit\* newDigit = new digit;

newDigit->value = value;

newDigit->next = NULL;

if (this->tail) {//if BigInt is !empty

newDigit->prev = tail;

tail->next = newDigit;

}

else {

newDigit->prev = NULL;

this->head = newDigit;

}

this->tail = newDigit;

}

BigInt operator+(const BigInt& a, const BigInt& b) {

digit\* currentDigitA = a.head, \* currentDigitB= b.head;

BigInt sum;

int digitsSum, leftover = 0;

if (!a.head || !b.head)

sum = sum;//NaN

else if ((absVal(a) <=> (BigInt)"0") == 0)//a=0

sum = b;

else if (((absVal(b) <=> (BigInt)"0") == 0))//b=0

sum = a;

else if ((a.isPositive && b.isPositive) || (!a.isPositive && !b.isPositive)) {//a+b or -(a+b)

while (currentDigitA || currentDigitB) {//|a|+|b|//default case

digitsSum = leftover;

if (currentDigitA)

digitsSum += currentDigitA->value;

if (currentDigitB)

digitsSum += currentDigitB->value;

leftover = 0;

while (digitsSum > 9) {

digitsSum -= 10;

leftover++;

}

sum.addToTail(digitsSum);

if(currentDigitA)

currentDigitA = currentDigitA->next;

if(currentDigitB)

currentDigitB = currentDigitB->next;

}

if (leftover > 0)

sum.addToTail(leftover);

sum.isPositive = a.isPositive;//optionally \*-1

}

else if ((a.isPositive && !b.isPositive))//a-b

sum = a - absVal(b);

else if (!a.isPositive && b.isPositive)//-(a-b) = b-a

sum = b - absVal(a);

return sum;

}

BigInt BigInt::operator+() const {

BigInt positiveThis;

positiveThis = \*this;

return positiveThis;

}

BigInt BigInt::operator-() const {

BigInt negativeThis;

negativeThis = \*this;

negativeThis.isPositive = !negativeThis.isPositive;

return negativeThis;

}

BigInt operator-(const BigInt& a, const BigInt& b) {

digit\* currentDigitA = a.head, \* currentDigitB = b.head;

BigInt difference;

int digitsDifference, credit = 0;

if (!a.head || !b.head)

difference = difference;//already NaN

else if ((absVal(a) <=> (BigInt)"0") == 0)

difference = -b;

else if (((absVal(b) <=> (BigInt)"0") == 0))

difference = a;

else if ((a.isPositive && b.isPositive) || (!a.isPositive && !b.isPositive)) {//a-b or -(a-b)=b-a

if ((absVal(a) <=> absVal(b)) >= 0) {//a>=b

while (currentDigitA) {

digitsDifference = credit;

digitsDifference += currentDigitA->value;

if (currentDigitB)

digitsDifference -= currentDigitB->value;

credit = 0;

while (digitsDifference < 0) {

digitsDifference += 10;

credit--;

}

difference.addToTail(digitsDifference);

if (currentDigitA)

currentDigitA = currentDigitA->next;

if (currentDigitB)

currentDigitB = currentDigitB->next;

}

difference.isPositive = a.isPositive;

}

else if ((absVal(a) <=> absVal(b)) == 0)

difference = "0";

else//a<b

difference = - (b - a);

}

else if((a.isPositive && !b.isPositive) || (!a.isPositive && b.isPositive)){

difference = absVal(a) + absVal(b);

difference.isPositive = a.isPositive;

}

while ((difference.tail != difference.head) && difference.tail->value == 0)

difference.deleteTail();

return difference;

}

void BigInt::deleteDigits() {

digit\* currentDigit = this->head, \*deletable;

while (currentDigit) {

deletable = currentDigit;

currentDigit = currentDigit->next;

delete deletable;

}

this->head = NULL;

this->tail = NULL;

}

BigInt absVal(const BigInt& a) {

BigInt absolute;

absolute = a;

absolute.isPositive = true;

return absolute;

}

BigInt& BigInt::operator=(const BigInt& rhs) {

if (&rhs != this) {

this->deleteDigits();

if (BigIntToStr(rhs) != "NaN")

\*this = BigIntToStr(rhs);

}

return \*this;

}

BigInt::BigInt(const BigInt& rhs) {

this->head = this->tail = NULL;

\*this = rhs;

}

BigInt::BigInt(const std::string& rhs) {

this->head = this->tail = NULL;

\*this = rhs;

}

BigInt operator\*(const BigInt& a, const BigInt& b) {

BigInt product, operand1, operand2, zero, one;

product = zero = "0";

one = "1";

if (!a.head || !b.head)

product = operand1;//NaN

else if (((a <=> zero) == 0) || ((b <=> zero) == 0))

product = product;//0

else if ((a <=> zero) < 0) {

product = absVal(a) \* b;

product.isPositive = !product.isPositive;

}

else if ((b <=> zero) < 0) {

product = a \* absVal(b);

product.isPositive = !product.isPositive;

}

else {//a,b>0

operand1 = a;

operand2 = b;

if (operand1 > operand2) {

BigInt tmp = operand2;

operand2 = operand1;

operand1 = tmp;

}

for (operand1; (operand1 <=> zero) > 0; operand1 = operand1 - one)

product = product + operand2;

}

return product;

}

void BigInt::deleteTail() {

if (tail) {

if (head == tail) {

delete head;

head = tail = NULL;

}

else {

digit\* tmp = tail->prev;

delete tail;

tmp->next = NULL;

tail = tmp;

}

}

}

BigInt operator/(const BigInt& a, const BigInt& b) {

BigInt quotient, newA, one, zero;

quotient = zero = "0";

one = "1";

if (!a.head || !b.head)

quotient = newA;//NaN

else if ((absVal(b) <=> zero) == 0)

quotient = newA;//NaN

else if ((absVal(a) <=> zero) == 0)

quotient = quotient;//already zero

else if ((a <=> zero) < 0) {

quotient = absVal(a) / b;

quotient.isPositive = !quotient.isPositive;

}

else if ((b <=> zero) < 0) {

quotient = a / absVal(b);

quotient.isPositive = !quotient.isPositive;

}

else {

newA = a;

while ((newA <=> b) >= 0) {//a,b >0

newA = newA - b;

quotient = quotient + one;

}

}

return quotient;

}

BigInt::~BigInt() {

this->deleteDigits();

}

# **7. Результати тестування програми та їх аналіз.**

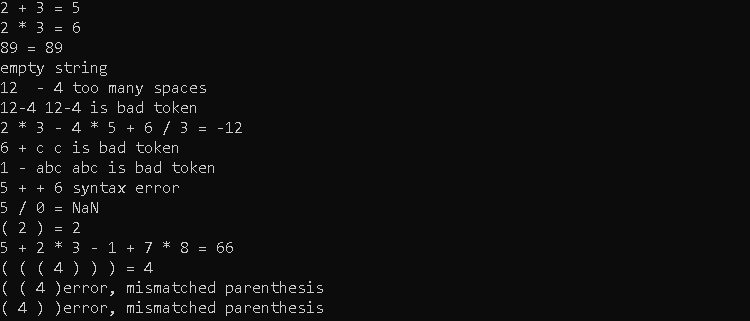


Рисунок 2

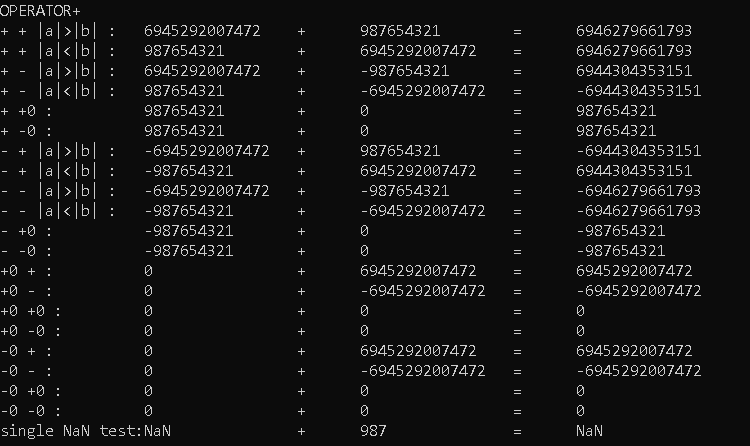


Рисунок 3

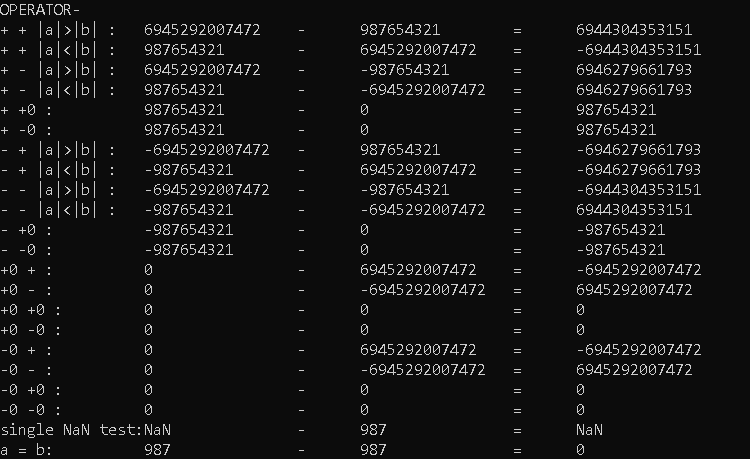


Рисунок 4

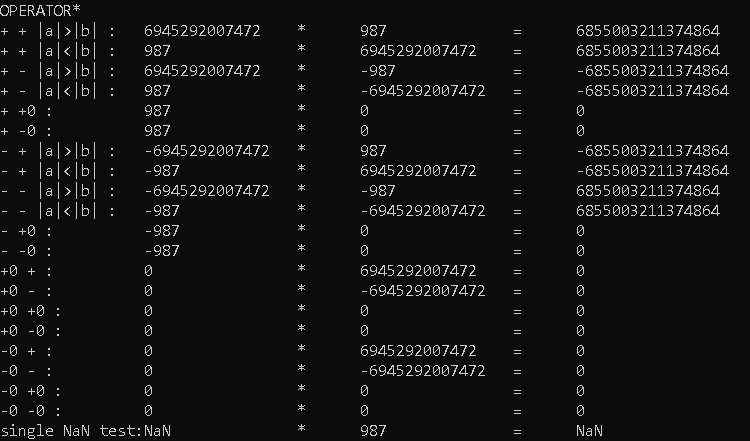


Рисунок 5

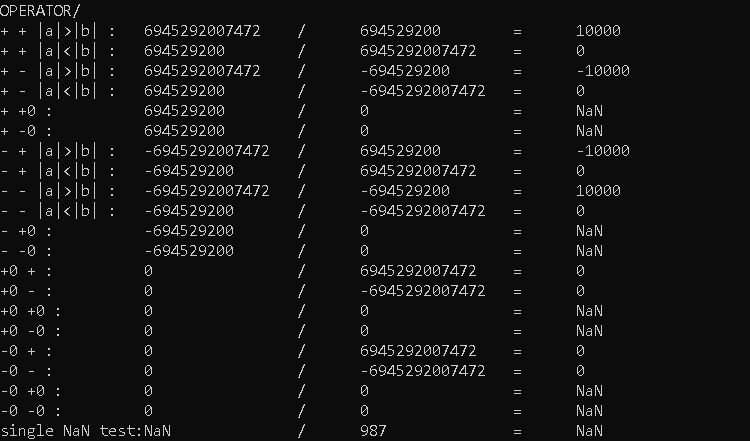


Рисунок 6

Результати відповідають очікуванням, програма працює коректно.

# **8. Висновки щодо використання лінійних динамічних структур.**

Лінійні динамічні структури – масиво подібні об’єкти, розмір яких можна міняти у реальному часі. Лінійні тому що можна мати доступ до усіх елементів через один. Вони розділяються на два основні типи: однозв’язні та двузв’язні, в перших рухатись по елементам можна лише в одну сторону, тож доцільно зберігати тільки початок списку(за деякими винятками, коли потрібно дуже часто звертатися до останнього елемента, або додавати новий, насправді це буває дуже часто), в двузв’язних можна рухатись в обидві сторони, тож зберігати і кінець і початок майже обов’язково. Також вони можуть бути зацикленими – тоді достатньо зберігати вказівник на будь який елемент, бо навіть в однозв’язному можна буде дійти до кожного елементу.

Такі структури дуже зручно використовувати для роботи з даними, які не мають фіксованого розміру, це зручніше ніж std::vector, бо вектор завжди намагається виділити послідовну пам’ять, та через те що пам’ять виділяється достатньо рандомно, може виникнути ситуація, коли вектору для додавання нового елементу потрібно “переїхати” і вільна пам’ять існує, але він не може її використати через те, що не має послідовного доступу до неї. В той самий час динамічні списки можуть застосувати цю пам’ять і можна навіть реалізувати індексацію.