МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

другий (магістерський)

(рівень вищої освіти)

Дослідження можливостей дерев рішень та методів синтаксичного аналізу арифметичних процесів для оптимізації роботи програм.

Виконав:
Студент II курсу, групи ІПЗм-21-2

Перетяга М. Ю
(ім'я, прізвище)

Спеціальність 121 Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо – наукова
(освітньо-професійна або освітньо – наукова)

Керівник проф. Лєсна Н.С.
(посада, ім'я, прізвище)

Допускається до захисту

Зав. Кафедри _____ З.В. Дудар

2023р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Програмної Інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва)

Тип програми освітньо-наукова програма

Освітня програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Перетязі Максиму Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи: Дослідження можливостей дерев рішень та методів синтаксичного аналізу арифметичних процесів для оптимізації роботи програм

затверджена наказом університету від «29» 03 2023 р. № 302Ст

2. Термін подання студентом роботи до екзаменаційної комісії «01» 05 2023 р.

3. Вихідні дані до роботи встановлений календарний план роботи, методичні вказівки до оформлення пояснювальної записки.

4. Перелік питань, що потрібно опрацювати в роботі аналіз предметної галузі, огляд наявних математичних моделей, аналіз існуючих алгоритмів підрахунку арифметичних виразів та аналіз відповідних структур даних.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз предметної галузі	10.11.2022	виконано
2	Постановка задачі	20.11.2022	виконано
3	Здійснення огляду математичних моделей	25.12.2022	виконано
4	Аналіз існуючих моделей прогнозування	30.01.2023	виконано
5	Дослідження обраних алгоритмів	01.03.2023	виконано
6	Написання пояснювальної записки	15.04.2023	виконано
7	Підготовка презентації та доповіді	25.04.2023	виконано
8	Перевірка роботи на плагіат та нормоконтроль	12.05.2023	виконано
9	Захист кваліфікаційної роботи	20.05.2023	виконано

Дата видачі завдання 29 березня 2023 р.

Студент _____
(підпис)

Керівник роботи _____ _____проф. Лєсна Н.С._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Кваліфікаційна робота магістра містить: 82 с., 6 рис., 7 табл., 5 додатків, 11 джер.

ДЕРЕВА РІШЕНЬ, СИНТАКСИЧНИЙ АНАЛІЗ АРИФМЕТИЧНИХ ПРОЦЕСІВ, ОПТИМІЗАЦІЯ.

Предметом дослідження є методи синтаксичного аналізу арифметичних процесів дерева рішень.

Метою роботи є проведення дослідження методів синтаксичного аналізу арифметичних процесів дерева рішень для оптимізації програм.

У результаті роботи була здійснена підготовча робота для подальшого дослідження та розроблена документація для майбутньої системи оптимізації програм.

EXPRESSION TREES, SYNTAX ANALYSIS OF ARITHMETIC PROCESSES, OPTIMIZATION.

The subject of the research is methods of syntactic analysis of arithmetic processes of expression trees.

The purpose of the work is to conduct research on methods of syntactic analysis of arithmetic processes of expression trees for program optimization.

As a result of the work, preparatory work for further research was carried out and documentation was developed for the future optimization system.

Умови публікації пояснювальної записки

Я, Перетяга Максим Юрійович
(прізвище, ім’я, по батькові)
студент групи ІПЗм-21-2 здобувач вищої освіти на другому (магістерському) рівні

кафедра програмної інженерії
(повна назва кафедри)

заявляю: моя кваліфікаційна робота на тему «Дослідження можливостей дерев рішень та методів синтаксичного аналізу арифметичних процесів для оптимізації роботи програм», що буде представлена до ЕК для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлен з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

З давніх часів людину цікавили завдання, пов'язані з питанням визначення найкращого (оптимального) варіанта рішення.

Оптимізація – процес приведення системи у найкращий (оптимальний) стан.

В інформатиці, оптимізація програми або оптимізація програмного забезпечення є процесом модифікації програмної системи, щоб змусити її працювати більш ефективно або використовувати менше ресурсів. Загалом комп'ютерна програма може бути оптимізована так, щоб вона виконувалася швидше, або щоб вона могла працювати з меншим об'ємом пам'яті або іншими ресурсами, або споживати менше енергії.

Пошук нових методів оптимізації програмних систем був і завжди буде актуальним у сфері інформаційних технологій. Одним із можливих методів може бути оптимізація застосунку завдяки представленню арифметичних виразів у вигляді дерева рішення.

Дерево рішень – метод представлення вирішальних правил у певній ієрархії, що включає елементи двох типів - вузлів (node) і листя (leaf). Вузли включають вирішальні правила і проводять перевірку прикладів на відповідність обраного атрибуту навчальної множини.

Метою дипломної роботи є дослідження методів синтаксичного аналізу арифметичних виразів, а також аналіз існуючих структур даних, та вибір найбільш підходящу структуру, яка зможе зберігати набори арифметичних виразів у ієрархічній формі.

1 ОПИС ПРОБЛЕМНОЇ ГАЛУЗІ

1.1 Аналіз предметної області

Арифметичні вирази є надзвичайно важливими у фундаментальному синтаксисі комп'ютера, оскільки вони надають числові значення, які підтримують функції коду. Навпаки, інші види виразів, такі як символьні або логічні вирази, містять різні види індикаторів.

Символьні вирази містили текстові значення або окремі літери або символи для аналізу або відображення, тоді як логічні вирази містять одне з двох логічних значень: true або false.

Існує два види арифметичних виразів у синтаксисі комп'ютерного програмування: цілі чи дійсні числа та дійсні числа чи числа з плаваючою комою. Останні використовуються для ідентифікації та зберігання комплексних чисел, які можуть не вписатися в ціле значення.

Константи, змінні, покажчики функцій, і навіть висловлювання у круглих дужках називаються операндами. Арифметичний вираз може бути у правій частині оператора присвоєння, у списку фактичних параметрів при зверненні до підпрограми. Арифметичні висловлювання використовуються під час запису відносин.

У комп'ютерному коді оператори та функції працюють з окремими виразами або наборами виразів, включаючи арифметичні, символьні та логічні вирази. Вони забезпечують основу для видів обробки даних, які виконуються у програмному забезпеченні.

Знаки арифметичних операцій:

- + (плюс) додавання;
- - (мінус) віднімання;
- * (зірочка) множення;
- / (слэш) поділ;
- ** (дві зірочки) зведення в ступінь.

Дерево виразів — це представлення виразів, упорядкованих у деревоподібній структурі даних. Іншими словами, це дерево з листками як операндами виразу, а вузли містять оператори. Подібно до інших структур даних, взаємодія даних також можлива в дереві виразів. Дерева виразів(див. рис. 1.1) в основному використовуються для аналізу, оцінки та модифікації виразів, особливо складних виразів.

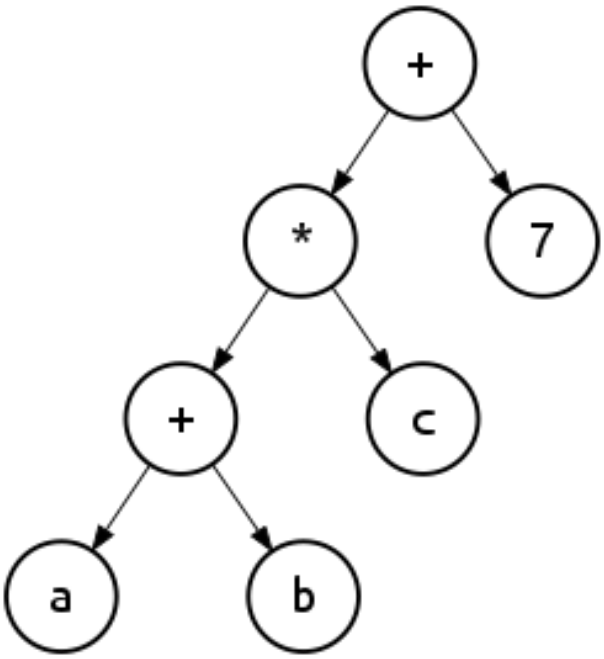

Рисунок 1.1 – Дерево виразів (виконано самостійно)

Дерева виразів є одним із найкращих методів представлення коду рівня мови у формі даних, які зберігаються в структурі у формі дерева. Дерево виразів вважається представленням лямбда-виразу в пам’яті. Дерево робить структуру, що містить лямбда-вираз, більш явною та прозорою. Дерево виразів було створено для перетворення коду в рядок, який можна передати іншим процесам як вхідні дані. Він містить фактичні елементи, задіяні в запиті, а не фактичний результат запиту.

Одна з важливих властивостей дерев виразів полягає в тому, що вони є незмінними, тобто для того, щоб змінити існуюче дерево виразів, необхідно побудувати нове дерево виразів шляхом копіювання та зміни існуючого виразу дерева. Коли справа доходить до програмування, дерево виразів зазвичай будується з постфіксними виразами, у яких зчитується один символ за раз. Якщо символ є

операндом, створюється дерево з одним вузлом, а вказівник на нього поміщається в стек.

1.2 Постановка задачі

З самого початку існування комп’ютерних наук розробники стикалися з проблемою, коли виконання програмного коду займає більше часу ніж очікувалось. Тому проблема оптимізації програм завжди є і буде актуальною. Люди завжди будуть шукати шляхи як можна прискорити їх програмний застосунок, використовуючи різні методи, та створюючи нові алгоритми.

Оптимізацією програми називають такі перетворення, які дозволяють зробити її ефективнішою, тобто, зробити її більш економною по пам'яті та/або швидше за виконання тих же функцій, що і до оптимізаційного перетворення.

Два приватних критерії оптимізації - час виконання програми та обсяг використовуваної нею пам'яті, у випадку, суперечать одне одному, як і ефективне написання програм супроводжується збільшенням роботи програміста. Відомо, що скорочення часу виконання програми, як правило, можна домогтися за рахунок збільшення обсягу використовуваної пам'яті і навпаки. У цьому випадку при виборі необхідного критерію набувають чинності евристичні міркування програміста, що віддає перевагу одному з них. Зазвичай конкретні обставини диктують важливість оптимізації програми за часом чи пам'яті.

Є багато різних способів оптимізації та прискорення праці програмного коду, використовуючи різні підходи та алгоритми. Одним із таких є методи синтаксичного розбору арифметичних виразів на операції та представлення у виді наборів операцій у деревовидній структурі даних, а сам у виді дерева рішень.

Задачею цієї роботи є ознайомлення та дослідження дерева рішень як структуру даних яка має можливість зберігати ієрархічну послідовність операцій арифметичних виразів, а також аналіз алгоритмів синтаксичного парсингу арифметичних виразів з метою оптимізувати існуючі алгоритми.

Дерево рішень є одним з ефективних інструментів для оптимізації програм, які містять арифметичні вирази. Його основна ідея полягає в тому, що він представляє вираз у вигляді дерева, де вузлами є оператори, а листямі - операнди. Така структура дозволяє з легкістю виконувати операції з виразами, такі як обчислення, зведення відносин та оптимізації.

Алгоритм синтаксичного парсингу є процесом, який перетворює вихідний код в програму, що може бути виконана комп’ютером. Цей алгоритм включає в себе такі етапи як лексичний аналіз, синтаксичний аналіз та семантичний аналіз. В процесі парсингу, дерево рішень будується з використанням правил граматики, яка описує синтаксис мови програмування.

З використанням дерева рішень можна зробити оптимізацію програмного коду, що містить арифметичні вирази, шляхом перетворення його у більш оптимізовану форму. Наприклад, дерево рішень дозволяє проводити операції з виразами у більш оптимальному порядку, що може зменшити час виконання програми.

Окрім того, існують різні алгоритми синтаксичного парсингу арифметичних виразів, які мають свої переваги та недоліки з точки зору часу виконання та складності. Тому, дослідження та оптимізація алгоритмів синтаксичного парсингу є важливою задачею для розробників програмного забезпечення.

2 МАТЕМАТИЧНЕ ПРЕДСТАВЛЕННЯ

2.1 Аналіз існуючих структур даних придатних для зберігання арифметичних операцій

Вирішимо та промоделюємо векторну оптимізаційну задачу:

Обрати якомога кращу, ефективну та гнучку структуру даних для зберігання набору арифметичних операцій зі швидким читанням даних, швидким записом даних, легким у написанні синтаксисом, оптимальною за використанням ресурсних даних, тобто легковажне та інтеграційність.

Проведемо інформаційну підготовку прийняття рішення з вибору необхідної структури даних (див. табл. 1):

- опишемо множину альтернатив:
  1) linked list;
  2) stack;
  3) array;
  4) expression tree;
  5) queue.
- опишемо критерії вибору:
  1) швидкість читання даних – час, за який виконується зчитування значення з обраної структури даних;
  2) швидкість запису даних – час, за який виконується запис значення з обраної структури даних;
  3) простота реалізації – наскільки зручний у написанні синтаксис, та скільки коду потрібно написати для реалізації програми;
  4) легковажність – кількість ресурсних даних потрібно на запуск і працю програми;
  5) гнучкість - можливість редагувати набір арифметичних операцій та її послідовність;
  6) інтеграційність – складність та можливість інтеграції з сторонніми бібліотеками.

– опишемо шкали оцінок за критеріями:

1) швидкість запису, швидкість читання може бути оцінено у складності алгоритму;

2) простота реалізації, легковажність, інтеграційність – кількісні показники які матимуть шкалу оцінювання від 1 до 5, 1 – це найвища оцінка, а 5 – найнижча.

Моделювання задачі

Розглянемо дані показників у таблиці 2.1.

Таблиця 2.1 - Модель

Показник	Вид структури даних				
	Linked list	Stack	Array	Expression Tree	Queue
Швидкість читання	O(n)	O(1)	O(1)	O(log n)	O(1)
Швидкість запису	O(1)	O(1)	O(1)	O(log n)	O(1)
Простота реалізації	4	4	5	2	3
Легковажність	2	2	2	3	2
Інтеграційність	5	5	5	1	5

Змінимо модель задачі для подальшого аналізу наступним чином: швидкість читання та швидкість запису переведемо у 5ти бальну систему, де O(1) – 1, O(log n) – 2, O(n) – 3.

Нову модель бачимо нижче (див. табл. 2.2):

Таблиця 2.2 – Модифікована модель

Показник	Вид структури даних				
	Linked list	Stack	Array	Expression Tree	Queue
Швидкість читання	3	1	1	2	1

Продовження таблиці

Швидкість запису	1	1	1	2	1
Простота реалізації	4	4	5	2	4
Легковажність	2	2	2	3	2
Інтеграційність	5	5	5	1	5

На даному етапі мови порівняльні за принципом Парето, де ми бачимо, що можна закреслити мову програмування Linked list та Array (див. табл. 2.3).

Таблиця 2.3 – Порівняльна таблиця за Парето

Показник	Вид структури даних				
	Linked list	Stack	Array	Expression Tree	Queue
Швидкість читання	3	1	1	2	1
Швидкість запису	1	1	1	2	1
Простота реалізації	4	4	5	2	4
Легковажність	2	2	2	3	2
Інтеграційність	5	5	5	1	5

Проведемо лінійну адитивну згортку з нормуючими множниками для усієї моделі задачі (див. табл. 2.4). Додаємо коефіцієнти, до параметру Інтеграційність з вагою 3, усі інші остаються з вагою 1, тому що інтеграційність в пріоритеті, алгоритм повинен бути легко використовуватися з іншими бібліотеками, наприклад з ORM бібліотеками, які будуть вживати набір операцій для того щоб згенерувати оптимальний запит до бази даних.

Таблиця 2.4 – Згорткова модель

Показник	Вид структури даних			
	Stack	Expression Tree	Queue	Нормуючий множник
Швидкість читання	1	2	1	0.25
Швидкість запису	1	2	1	0.25

Продовження таблиці

Простота реалізації	4	2	4	0.1
Легковажність	2	3	2	0.142857
Інтеграційність	5	1	5	0.111111
Результати	2.185714286	1.961904762	2.852380952	

Як було зазначено у роботі чим нижче оцінка, тим краще значення характеристики, тому обираємо структуру даних з найменшим значенням лінійної згортки – Expression Tree.

2.2 Опис обраної моделі

Expression Tree – це дерево, яке використовується для представлення різних виразів. Деревоподібна структура даних використовується для представлення виразів. У цьому дереві внутрішній вузол завжди означає оператори.

Особливості дерева виразу:

- листові вузли завжди позначають операнди;
- операції завжди виконуються над цими операндами;
- оператор, який є у глибині дерева, завжди має найвищий пріоритет;
- оператор, що не сильно на глибині дерева, завжди має менший пріоритет у порівнянні з операторами, що лежать на глибині;
- операнд завжди буде у глибині дерева; отже він вважається найвищим пріоритетом серед усіх операторів;
- основне використання дерев виразів полягає в тому, що вони використовуються для оцінки, аналізу та зміни різних виразів;
- вони також використовується для з'ясування асоціативності кожного оператора у виразі;
- дерева виразів формуються з допомогою контекстно-вільної граматики.
- основною метою використання дерев виразів є створення складних виразів, які можна легко обчислити за допомогою цих виразів;

– воно незмінне, і після того, як ми створили дерево виразів, ми не можемо змінити його чи модифікувати далі;
– щоб зробити додаткові зміни, необхідно повністю побудувати нове дерево виразів.

Дерева виразів відіграють дуже важливу роль у поданні коду рівня мови у вигляді даних, які переважно зберігаються в деревоподібній структурі. Він також використовується у поданні пам'яті лямбда-виразу. Використовуючи деревоподібну структуру даних, ми можемо виразити лямбда-вираз прозоріше і явно. Спочатку він створюється для перетворення сегмента коду на сегмент даних, щоб вираз можна було легко обчислити.

Дерево виразу є двійкове дерево, в якому кожен зовнішній або листовий вузол відповідає операнду, а кожен внутрішній або батьківський вузол відповідає операторам, тому, наприклад, дерево виразу (рис. 2.1) для 7 + ((1+8)*3) буде таким:

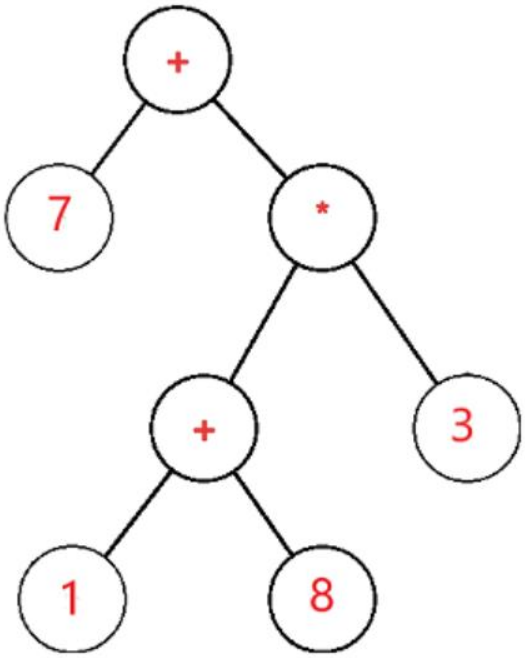

Рисунок 2.1 – Приклад представлення виразу у дереві (виконано самостійно)

Далі наведено приклад простої програмної реалізації. Створимо клас ExpressionTreeNode, який буде представляти вузол дерева, в нього будуть послання на дітей, а також буде зберігатися данні вузла (рисунок 2.2).

```
public class ExpressionTreeNode
{
    8 usages
    public double? Value { get; set; }
    6 usages
    public bool IsOperator { get; set; }
    33 usages
    public string? Operator { get; set; }
    17 usages
    public ExpressionTreeNode? Left { get; set; }
    11 usages
    public ExpressionTreeNode? Right { get; set; }
}
```

Рисунок 2.2 – Вузол дерева (виконано самостійно)

У даному коді представлена модель вузла для дерева рішень. Вузол містить наступні поля:

- value – значення вузла, тип даних - double. Якщо вузол є оператором, то це значення буде містити порожнє значення (null);
- isOperator – булеве значення, що вказує, чи є даний вузол оператором. Якщо значення - true, то це означає, що вузол є оператором, в іншому випадку - це операнд;
- operator – рядок, що містить знак операції, якщо вузол є оператором. У випадку, якщо вузол є операндом, то це значення буде містити порожній рядок (null);
- left – лівий нащадок вузла типу ExpressionTreeNode. Він може бути порожнім (null), якщо вузол є листом або якщо має тільки одного нащадка;
- right – правий нащадок вузла типу ExpressionTreeNode. Він може бути порожнім (null), якщо вузол є листом або якщо має тільки одного нащадка.

Ця модель вузла дерева рішень дозволяє зберігати ієрархічну структуру арифметичного виразу, яка складається з операторів та операндів. Дерево рішень будується з використанням даної моделі вузла, де кожен вузол може бути оператором або операндом, а також містить лівого та правого нащадки, що дозволяє зберігати послідовність операцій та їх відносні пріоритети.

Само дерево має дуже просту структуру і має вигляд (рисунок 2.1). Цей код представляє клас ExpressionTree з властивістю Root типу ExpressionTreeNode. Властивість Root представляє кореневий вузол дерева рішень, яке може бути створене за допомогою класу ExpressionTreeNode.

Символ питання після типу даних означає, що ця змінна може бути null. Тому Root може мати значення null, якщо дерево рішень ще не було створено або якщо воно було знищене.

Клас ExpressionTree зазвичай використовується для представлення дерева рішень, яке містить арифметичний вираз, що може бути обчислений. Кожен вузол дерева рішень, що містить число або оператор, представлений класом ExpressionTreeNode, який містить додаткові властивості для зберігання інформації про вузол, такі як тип оператора, ліва і права гілки тощо (див. рис. 2.3).

```
public class ExpressionTree
{
    33 usages
    public ExpressionTreeNode? Root { get; set; }
}
```

Рисунок 2.3 – Дерево (виконано самостійно)

Дерево рішень найбільше підходить для зручного використання арифметичних виразів, для того щоб зберігати усю ієрархічну структуру алгоритму, розбитих на операнди та цифри.

Також у якості проміжної структури даних буде використовуватися Stack, це базова структура яка використовується у дуже великій кількості алгоритмів.

Stack (стек) – це структура даних, яка працює за принципом "останній прийшов - перший вийшов (рисунок 2.4).

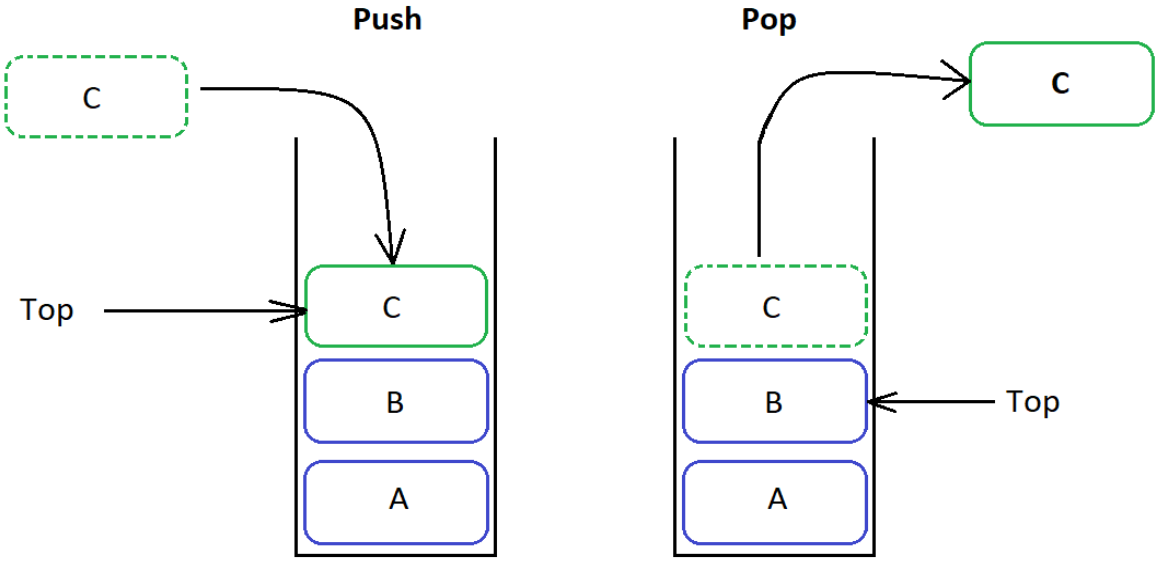

Рисунок 2.4 – Принцип дії Stack структури даних (виконано самостійно)

Використання стеку в поєднанні з деревом рішень дозволяє зберігати проміжні результати обчислень та оптимально організовувати процес виконання арифметичних операцій, що зменшує навантаження на процесор та пам'ять. Зокрема, під час обходу дерева рішень для виконання арифметичних операцій, можна використовувати стек для зберігання тимчасових значень, які необхідні для обчислення виразів, що знаходяться на нижніх рівнях дерева. Таким чином, використання дерева рішень та стеку дозволяє ефективно виконувати складні арифметичні операції та зберігати ієрархічну структуру алгоритму.

3 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТУ

Основна мета цього експерименту — порівняти продуктивність алгоритму на основі дерева виразів з іншими поширеними методами обчислення арифметичних виразів.

Окрім порівняння ефективності різних методів, мета експерименту – оцінити ефективність кожного методу в обробці виразів різного розміру та складності. Це також дасть уявлення про ефективність кожного методу в роботі з основними арифметичними операціями, а також більш складними операціями, такими як зведення в ступінь, логарифми та тригонометричні функції. Крім того, результати експерименту будуть корисними для вибору найбільш прийнятного методу для додатків, які передбачають оцінку арифметичних виразів, наприклад у мовах програмування чи наукових обчисленнях. Зрештою, експеримент має на меті зробити внесок у сукупність знань про ефективність і точність методів обчислення арифметичних виразів.

3.1. Вхідні дані

Створіть набір арифметичних виразів для обчислення. Вирази мають бути згенеровані випадковим чином і мати різні розміри та складність. Вхідні дані мають включати вирази лише з основними арифметичними операціями (додавання, віднімання, множення, ділення) та вирази зі складнішими операціями (зведення в ступінь, логарифми, тригонометричні функції тощо). Вхідні вирази мають бути представлені в інфіксній нотації (рисунок 3.1).

```
(4 + 6) * 7 - 5
sin(30) + sqrt(9) / log10(100)
2 * (5 + 6) / 3
2^3 + 4 * log(100) - 5
cos(45) * tan(60) + 3^4 - 8
1 + 2 * 3 / 4 - 5^2
sqrt(16) + 4^(1/2) / log(1000) * 5
tan(45) + cos(30) - sin(60)
7 - 2 * (5 + 3) / (4 - 2^2)
log(100) + sin(45) * tan(60) - 6^2
3 / 2 * pi + sqrt(25) - log(1) * 8
sin(pi/2) * cos(pi/3) + 2^3 - 5
log10(10000) + tan(pi/4) - 3^2
sqrt(25) + (5 * 6) / (9 - 3) + 2
2^2 + 2 * 3^2 + 2 * 2 * 3
sin(pi/6) + cos(pi/4) - tan(pi/3) + 4^0
log(100000) - sqrt(64) * 2 + 3^3
3 * (4 + 2^2) / (5 - 1) - cos(pi/2)
2 * pi * sqrt(10) + 5^2 - log(1)
sqrt(4) + 4 * sin(pi/6) - 2 * cos(pi/3) + 3^2
```

Рисунок 3.1 – Приклад вхідних даних для експерименту (виконано самостійно)

Арифметичні вирази, створені для цього експерименту, повинні охоплювати широкий діапазон складності, щоб гарантувати, що продуктивність алгоритмів може бути перевірена в різних умовах. Вирази мають бути згенеровані випадковим чином, щоб уникнути будь-якої упередженості в процесі відбору. Основні арифметичні операції повинні бути включені у вирази, щоб перевірити ефективність алгоритмів у виконанні простих обчислень. Крім того, слід включити більш складні операції, такі як піднесення до ступеня, логарифми та тригонометричні функції, щоб оцінити здатність алгоритмів виконувати складніші обчислення. Діапазон розмірів виразів також слід змінювати, щоб перевірити масштабованість і стійкість алгоритмів. Загалом, згенеровані вирази мають відповідати реальним арифметичним обчисленням.

3.2. Вибір технології

Платформою, на якій буде розроблятися цей алгоритм, було обрано платформу .NET, а саме мову програмування C#. А саме, через декілька переваг.

По-перше, .NET надає надійний та ефективний набір бібліотек для обробки математичних операцій, включаючи підтримку комплексних чисел, тригонометричних функцій тощо. Ці бібліотеки дозволяють розробникам легко виконувати обчислення та маніпулювати значеннями, використовуючи високорівневий та інтуїтивно зрозумілий синтаксис.

Крім того, об'єктно-орієнтовані принципи проектування .NET дозволяють легко створювати багаторазовий і модульний код, що може бути особливо корисним у випадку алгоритму. За допомогою .NET розробники можуть створювати класи та функції, які інкапсулюють логіку алгоритму, що дозволяє повторно використовувати код і підвищує його ремонтопридатність.

Ще однією перевагою .NET є підтримка різних типів і структур даних, включаючи масиви і стеки. Ці структури даних необхідні для реалізації алгоритму, який передбачає зберігання та маніпулювання стеком значень та операторів. Також вже існуючі структури даних стануть у нагоді при створенні власного дерева виразів, яке ми будемо використовувати для зберігання математичних операторів та чисел.

Ще одна перевага платформи .NET полягає у вбудованій підтримці збору сміття та управління пам'яттю, що допоможе забезпечити ефективне використання системних ресурсів, що особливо важливо у випадку складних алгоритмів, подібних до нашого.

Нарешті, .NET є кросплатформенною технологією. Це дає можливість використовувати існуючий код та бібліотеки на різних платформах. Розробники можуть повторно використовувати код, написаний на .NET для Windows та інших платформ, без необхідності переписувати його з нуля. Це спрощує розробку додатків, які безперешкодно працюють на різних платформах і забезпечують узгоджений користувацький досвід.

Таким чином, середовище програмування .NET пропонує різноманітні функції та інструменти, які роблять його добре придатним для реалізації алгоритму. Підтримка математичних операцій, об'єктно-орієнтовані принципи проектування, структури даних та управління пам'яттю роблять його потужною та ефективною платформою для розробки складних алгоритмів.

Також для проведення експерименту з дослідження можливостей дерев рішень та методів синтаксичного аналізу арифметичних процесів необхідно мати потужне обчислювальне обладнання, що забезпечить швидку обробку великої кількості даних.

Для цієї мети було обрано ноутбук Lenovo ThinkPad G2, який відповідає всім потребам експерименту. Даний ноутбук оснащений процесором Intel Core i7 8-го покоління з 4 ядрами і 8 потоками, що забезпечує достатню продуктивність та швидкість обробки даних для проведення експерименту. Крім того, ноутбук має достатній обсяг оперативної пам'яті та накопичувача, що дозволяє зберігати велику кількість даних без необхідності їх постійного копіювання.

Обрання Lenovo ThinkPad G2 як обладнання для проведення експерименту було здійснено на основі його технічних характеристик, доступності та вартості, що дозволяє забезпечити ефективність та якість проведення дослідження.

3.3. Реалізація алгоритмів

Для цього експерименту було реалізовано три різні методи обчислення арифметичних виразів. Візуально порівняти вихідні дані та виміряти час виконання всіх 3 алгоритмів, щоб визначити, який з них є найбільш оптимальним. Ми будемо використовувати наступні алгоритми:

- алгоритм створений на основі дерева виразів;
- (RPN) Reverse Polish Notation;
- алгоритм Recursive Descent.

3.3.1 Алгоритм створений на основі дерева виразів

Алгоритм на основі дерева використовує дерево виразів для оптимізації під виразів і підвищення ефективності, а також надає гнучкість модифікації арифметичних виразів, шляхом оновлення дерева, без повного перетворення. Алгоритм включає побудову дерева виразів з інфіксної нотації, а потім обчислення дерева для отримання результату. Алгоритм оптимізує процес оцінки, зменшуючи кількість зайвих обчислень, які необхідно виконати.

```
public static ExpressionTree ConvertToExpressionTree(string infix)
{
    var tokens = infix.Split(" ").ToList();
    var nodeStack = new Stack<ExpressionTreeNode>();
    var operatorStack = new Stack<string>();

    for (var index = 0; index < tokens.Count; index++)
    {
        var token = tokens[index];

        if (Token.IsNumber(token))
        {
            var value = double.Parse(token);
            var node = new ExpressionTreeNode
            {
                Value = value
            };

            nodeStack.Push(node);
        }
        else if (Token.IsConstant(token))
        {
            var node = new ExpressionTreeNode
            {
                Value = GetConstantValue(token)
            };

            nodeStack.Push(node);
        }
        else if (Token.IsOperator(token))
        {
            while (operatorStack.Count > 0 &&
IsHigherPrecedence(operatorStack.Peek(), token))
            {
                BuildNode(operatorStack, nodeStack);
            }
            operatorStack.Push(token);
        }
        else if (Token.IsFunction(token))
        {
            if (index + 3 <= tokens.Count && IsNumber(tokens[index + 2]) &&
```

```
tokens[index + 3] == ")")
            {
                var node = new ExpressionTreeNode()
                {
                    IsOperator = true,
                    Operator = token,
                    Left = new ExpressionTreeNode
                    {
                        Value = double.Parse(tokens[index + 2])
                    }
                };

                if (index > 0 && tokens[index - 1] == "(")
                {
                    operatorStack.Pop();
                }

                nodeStack.Push(node);

                index += 3;
            }
            else
            {
                operatorStack.Push(token);
            }
        }
        else if (Token.IsLeftParenthesis(token))
        {
            operatorStack.Push(token);
        }
        else if (Token.IsRightParenthesis(token))
        {
            while (operatorStack.Count > 0 &&
!Token.IsRightParenthesis(operatorStack.Peek()))
            {
                BuildNode(operatorStack, nodeStack);

                if (operatorStack.Count <= 0 ||
!Token.IsLeftParenthesis(operatorStack.Peek()))
                {
                    continue;
                }

                operatorStack.Pop();

                if (operatorStack.Count > 0 &&
!Token.IsFunction(operatorStack.Peek()))
                {
                    break;
                }
            }
        }
    }

    while (operatorStack.Count > 0)
    {
        BuildNode(operatorStack, nodeStack);
    }
```

```
    return new ExpressionTree { Root = nodeStack.Pop() };
}
```

Даний код є методом класу ExpressionTree, який приймає рядок у вигляді інфіксної форми запису математичного виразу і повертає дерево рішень для цього виразу.

Спочатку вхідний рядок розбивається на токени за допомогою методу Split() та записується в список tokens. Також створюється стек nodeStack для збереження вузлів дерева та стек operatorStack для збереження операторів.

Далі проходиться цикл по всім токенам в списку tokens. Якщо токен є числом, то створюється новий вузол ExpressionTreeNode, який містить це число, та додається до стеку nodeStack.

Якщо токен є константою, то створюється новий вузол ExpressionTreeNode, який містить значення цієї константи, та додається до стеку nodeStack.

Якщо токен є оператором, то виконується while – цикл, який буде доки стек operatorStack не буде порожнім і оператор на вершині стеку має більший пріоритет за поточний токен. Під час цього циклу викликається метод BuildNode, який використовує верхній елемент стеку operatorStack для створення нового вузла ExpressionTreeNode з двома дочірніми вузлами, які взято зі стеку nodeStack, та додає його до nodeStack. Потім поточний токен додається до operatorStack.

Якщо токен є функцією, то перевіряється чи має вона коректну форму, тобто має 2 аргументи: лівій дужці та числу після нього, а наступний токен - права дужка. Якщо так, то створюється новий вузол ExpressionTreeNode, який має значення функції, лівий дочірній вузол з числом, яке йде після лівої дужки, та додається до стеку nodeStack. Якщо ж форма некоректна, то токен додається до operatorStack.

Якщо токен - "(", то він додається до стеку операторів.

Якщо токен - ")", то метод використовує цикл while, щоб побудувати вузли дерева до тих пір, поки не зустріне відкриваючу дужку "(". Це виконується шляхом виклику методу BuildNode для кожного оператора в стеку операторів, поки не буде досягнута відкриваюча дужка. Якщо на вершині стеку операторів не є відкриваюча

дужка, цей оператор видаляється зі стеку операторів. Якщо на вершині стеку операторів знаходиться функція, то побудова вузлів припиняється.

Нарешті, після обробки всіх токенів у виразі, метод продовжує будувати вузли дерева, використовуючи всі залишені оператори у стеку операторів, викликаючи метод BuildNode для кожного з них.

У кінці метод повертає новий об'єкт ExpressionTree з коренем дерева, який є вершиною стеку вузлів.

Сильною стороною алгоритму, який ми розробляємо, є можливість оновлення дерева рішень без його повного перестворення. Це дозволяє ефективно виконувати зміни в інфіксному виразі та оновлювати результат у дереві без зайвих обчислень.

Ця можливість є дуже корисною для оптимізації роботи програми, оскільки дозволяє не повторювати обчислення випадкових частин виразу при зміні лише деяких елементів. Метод UpdateExpressionTree дозволяє виконувати такі оновлення, перераховуючи лише ті вузли, які були змінені у вхідному виразі. Це дозволяє зекономити час та ресурси, що є особливо важливим для великих обчислювальних задач, де вираз може містити сотні чи тисячі вузлів дерева.

```
public static void UpdateExpressionTree(ExpressionTree tree, string
newInfix)
{
    var newTokens = newInfix.Split(" ").ToList();
    var oldTokens = GetInfixTokens(tree.Root);

    var commonLength = Math.Min(newTokens.Count, oldTokens.Count);
    int i;

    for (i = 0; i < commonLength; i++)
    {
        if (newTokens[i] != oldTokens[i])
        {
            break;
        }
    }

    if (i < commonLength && IsNumber(newTokens[i]))
    {
        UpdateNodeValue(tree.Root, oldTokens[i],
double.Parse(newTokens[i]));
        i++;
    }
```

```
    var operatorStack = new Stack<string>();
    var nodeStack = new Stack<ExpressionTreeNode>();

    for (; i < newTokens.Count; i++)
    {
        var token = newTokens[i];

        if (Token.IsNumber(token))
        {
            var value = double.Parse(token);
            var node = new ExpressionTreeNode
            {
                Value = value
            };

            nodeStack.Push(node);
        }
        else if (Token.IsConstant(token))
        {
            var node = new ExpressionTreeNode
            {
                Value = GetConstantValue(token)
            };

            nodeStack.Push(node);
        }
        else if (Token.IsOperator(token))
        {
            while (operatorStack.Count > 0 &&
IsHigherPrecedence(operatorStack.Peek(), token))
            {
                BuildNode(operatorStack, nodeStack);
            }
            operatorStack.Push(token);
        }
        else if (Token.IsFunction(token))
        {
            if (i + 3 <= newTokens.Count && IsNumber(newTokens[i + 2]) &&
newTokens[i + 3] == ")")
            {
                var node = new ExpressionTreeNode()
                {
                    IsOperator = true,
                    Operator = token,
                    Left = new ExpressionTreeNode
                    {
                        Value = double.Parse(newTokens[i + 2])
                    }
                };

                if (i > 0 && newTokens[i - 1] == "(")
                {
                    operatorStack.Pop();
                }

                nodeStack.Push(node);
```

```
                i += 3;
            }
            else
            {
                operatorStack.Push(token);
            }
        }
        else if (Token.IsLeftParenthesis(token))
        {
            operatorStack.Push(token);
        }
        else if (Token.IsRightParenthesis(token))
        {
            while (operatorStack.Count > 0 && operatorStack.Peek() != "(")
            {
                BuildNode(operatorStack, nodeStack);

                if (operatorStack.Count <= 0 || operatorStack.Peek() !=
"(")
                {
                    continue;
                }

                operatorStack.Pop();

                if (operatorStack.Count > 0 &&
!IsFunction(operatorStack.Peek()))
                {
                    break;
                }
            }
        }
    }

    while (operatorStack.Count > 0)
    {
        BuildNode(operatorStack, nodeStack);
    }

    tree.Root = nodeStack.Pop();
}
private static void UpdateNodeValues(ExpressionTreeNode node,
     Dictionary<ExpressionTreeNode, double> updatedValues)
{
    if (node == null)
    {
        return;
    }

    if (updatedValues.ContainsKey(node))
    {
        node.Value = updatedValues[node];
    }

    UpdateNodeValues(node.Left, updatedValues);
    UpdateNodeValues(node.Right, updatedValues);
}
```

Метод UpdateExpressionTree оновлює дерево рішень, що було побудоване раніше за допомогою методу BuildExpressionTree на основі нового інфіксного виразу.

Алгоритм починає з розбиття нового інфіксного виразу на токени та отримання списку токенів дерева рішень. Далі, використовуючи ці два списки, алгоритм знаходить найдовшу спільну префіксну послідовність токенів між двома списками. Це дозволяє знайти ту частину дерева рішень, яка залишається незмінною.

Далі, алгоритм перевіряє, чи наступний токен в новому списку токенів є числом, якщо так, то значення відповідного вузла дерева змінюється на це число. Це дозволяє оновлювати значення вузлів, що не змінилися, а лише змінилася їхнє значення.

Після цього алгоритм працює з новими токенами, що залишилися. Він використовує дві стеки: operatorStack і nodeStack. Він проходить через кожний токен та виконує дії залежно від того, яким типом токену він є: число, оператор, функція або дужка. Це дозволяє побудувати нове дерево рішень, використовуючи нові значення.

Метод UpdateNodeValues оновлює значення вузлів дерева рішень. Він проходиться по дереву рішень та перевіряє, чи є поточний вузол у словнику оновлених значень. Якщо так, то значення відповідного вузла змінюється на нове значення. Це дозволяє змінювати значення вузлів у дереві рішень після оновлення інфіксного виразу, що було побудовано раніше.

Після знаходження оновленого значення вузла дерева, метод UpdateExpressionTree переставляє зразу за ним будь-які оператори та операнди, які змінилися в новому інфіксному виразі. Всі оператори та операнди, що залишилися незмінними, залишаються на своїх місцях в старому дереві.

Далі метод починає проходити через новий інфіксний вираз, токен за токеном, та добудовує нове дерево вузлів. При цьому, якщо знаходиться число або константа, створюється новий вузол з відповідним значенням. Якщо ж знаходиться оператор, то перевіряється, чи має він більший пріоритет за будь-який з операторів,

що залишилися на стеку операторів. Якщо має – вузли на стеку операторів злиттяються з вузлами зі стеку вузлів.

Таким чином, нове дерево вузлів побудоване з частково старих вузлів, що залишилися незмінними, та частково нових вузлів, які були створені на основі нового інфіксного виразу.

Метод UpdateExpressionTree дозволяє оновлювати дерево рішень без повного перетворення, що значно економить час виконання. Замість того, щоб перетворювати інфіксний вираз у постфіксний та створювати нове дерево, що може зайняти багато пам'яті та викликати затримки, цей метод оновлює старе дерево, розумно використовуючи вузли, що залишилися незмінними. Це зроблено за допомогою аналізу інфіксного виразу та відповідного оновлення старих вузлів і побудови нових вузлів для частини виразу, що змінилася.

3.3.2 Алгоритм Reverse Polish Notation

Алгоритм RPN (Reverse Polish Notation) складається зі способу запису математичного виразу та його обчислення. Записується він у постфіксному вигляді, де оператори знаходяться після операндів. Алгоритм RPN виконується наступним чином:

- створюється порожній стек;
- вхідний вираз зчитується зліва направо;
- якщо символ є операндом, то він додається до стеку;
- якщо символ є оператором, то зі стеку вилучаються два останніх операнди, над якими виконується дія залежно від оператора, результат дії додається до стеку;
- процес повторюється до тих пір, поки не буде оброблено всі символи вхідного виразу;
- коли весь вираз оброблено, єдиним елементом, що залишився у стеці є результат обчислень.

Алгоритм RPN можна використовувати для обчислення складних математичних виразів, тому що він не потребує використання додаткових дужок для визначення порядку виконання операцій. Зокрема, він корисний при роботі з

великими виразами, де важко зорієнтуватися у використанні дужок та при обробці математичних виразів у програмах та калькуляторах.

Загалом, алгоритм RPN є ефективним та простим у використанні способом запису та обчислення математичних виразів.

Нижче представлено код реалізації цього алгоритму.

```
public static string InfixToPostfixString(string infixString)
{
    var queue = GetPostfixQueue(infixString);
    var builder = new StringBuilder();

    while (queue.Any())
    {
        if (queue.Count > 1)
        {
            builder.Append(queue.Dequeue() + " ");
        }

        if (queue.Count == 1)
        {
            builder.Append(queue.Dequeue());
        }
    }

    return builder.ToString();
}

public static Queue<string> GetPostfixQueue(string input)
{
    var outputQueue = new Queue<string>();
    var operandStack = new Stack<string>();
    var inputArray = input.Split(' ');

    foreach (var token in inputArray)
    {
        if (Token.IsNumber(token))
        {
            outputQueue.Enqueue(token);
            continue;
        }

        if (Token.IsConstant(token))
        {
            outputQueue.Enqueue(Token.ReplaceConstant(token));
            continue;
        }

        if (Token.IsLeftParenthesis(token) || Token.IsFunction(token))
        {
            operandStack.Push(token);
            continue;
        }
```

```
        if (Token.IsRightParenthesis(token))
        {
            while (!Token.IsLeftParenthesis(operandStack.Peek()))
            {
                outputQueue.Enqueue(operandStack.Pop());
            }

            operandStack.Pop();
            continue;
        }

        while (operandStack.Any() &&
               Token.IsGreaterPrecedence(operandStack.Peek(), token) &&
               Token.IsLeftAssociated(token))
        {
            outputQueue.Enqueue(operandStack.Pop());
        }

        operandStack.Push(token);
    }

    while (operandStack.Count > 0)
    {
        outputQueue.Enqueue(operandStack.Pop());
    }

    return outputQueue;
}
```

Метод InfixToPostfixString приймає на вхід рядок infixString, що містить математичний вираз в інфіксному записі. Він викликає метод GetPostfixQueue, який повертає чергу, що містить математичний вираз у постфіксному записі. Потім метод InfixToPostfixString послідовно витягує елементи з черги, додаючи їх в об'єкт StringBuilder і повертає отриманий рядок.

Метод GetPostfixQueue реалізує алгоритм переведення інфіксного запису математичного виразу в постфіксний запис. Він використовує чергу outputQueue для зберігання результату і стек operandStack для зберігання операторів. Спочатку рядок, що містить математичний вираз, розбивається на масив токенів inputArray. Далі, для кожного токена виконується одна з таких дій:

- якщо токен є числом, він додається в outputQueue;
- якщо токен є константою, його замінюють на відповідне йому значення і додають у outputQueue;

- якщо токен є відкривною дужкою або функцією, він додається в operandStack;
- якщо токен є закривною дужкою, всі оператори з operandStack витягуються і додаються в outputQueue, поки не буде знайдена відповідна відкривна дужка, сама відкриваюча дужка видаляється з operandStack;
- якщо токен є оператором, витягуються всі оператори з operandStack, що мають більший або рівний пріоритет, і додаються в outputQueue. Потім поточний оператор додається в operandStack.

Після обробки всіх токенів усі оператори, що залишилися в operandStack, витягуються і додаються в outputQueue.

У результаті виконання методу GetPostfixQueue повертається черга outputQueue, що містить математичний вираз у постфіксному записі.

3.3.3 Алгоритм Recursive Descent

Алгоритм Recursive Descent (рекурсивний спуск) – це метод синтаксичного аналізу тексту, який використовується для перевірки відповідності вхідного тексту заданому контекстно-вільному граматиці (CFG).

Алгоритм Recursive Descent перебирає вхідний текст, щоб відповідно до правил граматики побудувати дерево розбору вхідного тексту.

Алгоритм складається зі спеціальної процедури (функції), яка виконується для кожного нетермінала граматики. Процедура для кожного нетермінала рекурсивна викликається для кожного правила граматики, в якому дане правило відображається. Під час виконання процедури для кожного нетермінала, алгоритм розглядає вхідний текст та порівнює його з правилами граматики, поки не знайде відповідну правильну послідовність терміналів, яка відповідає нетерміналу. У такому випадку процедура повертається до свого виклику, та продовжується розбір решти вхідного тексту.

Алгоритм Recursive Descent є дуже ефективним для граматик з невеликою кількістю нетерміналів, оскільки його легко реалізувати та зрозуміти, а також не вимагає додаткової обробки або перетворення вхідного тексту.

Однак, алгоритм Recursive Descent має обмеження на роботу з ліворекурсивними граматиками, оскільки такі граматики можуть призвести до зациклення рекурсивних викликів, що приведе до падіння алгоритму. Також, для граматик з великою кількістю нетерміналів та правил, Recursive Descent може бути неефективним з точки зору швидкодії.

Нижче приведена програмна реалізація алгоритму:

```
public static class RecursiveDescentEvaluator
{
    private static string _input;
    private static int _position;
    private static Dictionary<string, double> _variables = new
Dictionary<string, double>();

    public static double Evaluate(string infix)
    {
        _input = infix.Replace(" ", "");
        _position = 0;
        _variables.Clear();

        var result = ParseExpression();

        return result;
    }

    private static  double ParseExpression()
    {
        var result = ParseTerm();

        while (_position < _input.Length)
        {
            var op = _input[_position];
            if (op != '+' && op != '-')
            {
                break;
            }

            _position++;
            var term = ParseTerm();
            if (op == '+')
            {
                result += term;
            }
            else
            {
                result -= term;
```

```
            }
        }

        return result;
    }

    private static double ParseTerm()
    {
        var result = ParseFactor();

        while (_position < _input.Length)
        {
            var op = _input[_position];
            if (op != '*' && op != '/' && op != '%' && op != '^')
            {
                break;
            }

            _position++;
            var factor = ParseFactor();

            switch (op)
            {
                case '*':
                    result *= factor;
                    break;
                case '/':
                    result /= factor;
                    break;
                case '%':
                    result %= factor;
                    break;
                case '^':
                    result = Math.Pow(result, factor);
                    break;
            }
        }

        return result;
    }

    private static double ParseFactor()
    {
        if (_position >= _input.Length) throw new Exception("Unexpected end
of input");

        var nextChar = _input[_position];

        if (nextChar == '-')
        {
            _position++;
            return -ParseFactor();
        }

        if (nextChar == '(')
        {
            _position++;
            var result = ParseExpression();
```

```
            if (_position >= _input.Length || _input[_position] != ')')
            {
                throw new Exception("Expected closing parentheses");
            }

            _position++;
            return result;
        }

        if (char.IsDigit(nextChar) || nextChar == '.')
        {
            var numString = "";
            while (_position < _input.Length &&
(char.IsDigit(_input[_position]) || _input[_position] == '.'))
            {
                numString += _input[_position];
                _position++;
            }

            if (double.TryParse(numString, out var num))
            {
                return num;
            }
            else
            {
                throw new Exception($"Invalid number: {numString}");
            }
        }

        if (char.IsLetter(nextChar))
        {
            var funcName = "";
            while (_position < _input.Length &&
char.IsLetter(_input[_position]))
            {
                funcName += _input[_position];
                _position++;
            }

            if (_position < _input.Length && _input[_position] == '(')
            {
                _position++;
                var args = new List<double> { ParseExpression() };

                while (_position < _input.Length && _input[_position] ==
',')
                {
                    _position++;
                    args.Add(ParseExpression());
                }

                if (_position >= _input.Length || _input[_position] != ')')
                    throw new Exception("Expected closing parentheses");

                _position++;

                switch (funcName.ToLower())
                {
```

```
                    case "sin":
                        if (args.Count != 1)
                        {
                            throw new Exception("sin function takes exactly
1 argument");
                        }

                        return Math.Sin(args[0]);
                    case "cos":
                        if (args.Count != 1)
                        {
                            throw new Exception("cos function takes exactly
1 argument");
                        }

                        return Math.Cos(args[0]);
                    case "tan":
                        if (args.Count != 1)
                        {
                            throw new Exception("tan function takes exactly
1 argument");
                        }

                        return Math.Tan(args[0]);
                    case "exp":
                        if (args.Count != 1)
                        {
                            throw new Exception("exp function takes exactly
1 argument");
                        }

                        return Math.Exp(args[0]);
                    case "ln":
                        if (args.Count != 1)
                        {
                            throw new Exception("ln function takes exactly
1 argument");
                        }

                        return Math.Log(args[0]);
                    case "log":
                        if (args.Count != 2)
                        {
                            throw new Exception("log function takes exactly
2 arguments");
                        }

                        return Math.Log(args[0], args[1]);
                    default:
                        throw new Exception($"Unknown function:
{funcName}");
                }
            }
            else
            {
                if (_variables.ContainsKey(funcName))
                {
                    return _variables[funcName];
```

```
                }
                else
                {
                    throw new Exception($"Undefined variable: {funcName}");
                }
            }
        }

        throw new Exception($"Unexpected character: {nextChar}");
    }
}
```

Метод Evaluate отримує рядковий параметр infix, який представляє математичний вираз в інфіксній системі числення, і повертає значення типу double, яке представляє результат виразу. Спочатку метод видаляє всі пробіли у вхідному рядку та ініціалізує деякі змінні, зокрема вхідний рядок, поточну позицію у вхідному рядку та словник змінних. Потім він викликає метод ParseExpression для розбору та обчислення виразу і повертає результат.

Метод ParseExpression – це рекурсивна функція, яка розбирає і обчислює вираз, що складається з одного або декількох термів, розділених операторами додавання або віднімання. Спочатку викликається метод ParseTerm для розбору першого доданка, а потім повторно зчитується наступний символ у вхідному рядку, поки не буде знайдено оператор, відмінний від + або -. Якщо оператор +, він викликає метод ParseTerm для розбору наступного доданка і додає результат до накопиченого до цього часу результату. Якщо оператор -, він робить те саме, але замість цього віднімає доданок. Метод повертає накопичений результат.

Метод ParseTerm - це ще одна рекурсивна функція, яка аналізує і обчислює доданок, що складається з одного або декількох множників, розділених операторами множення, ділення, піднесення до степеня або піднесення до степеня. Спочатку вона викликає метод ParseFactor для аналізу першого множника, а потім повторно зчитує наступний символ у вхідному рядку, поки не знайде оператор, відмінний від *, /, % або ^. Якщо оператор *, вона викликає метод ParseFactor для аналізу наступного множника і множить отриманий результат на накопичений до цього часу результат. Якщо оператор /, він робить те ж саме, але замість цього ділить результат. Якщо оператор дорівнює %, він бере модуль результату з наступним множником. Якщо оператор ^, то результат підноситься до степеня

наступного множника за допомогою методу Math.Pow. Потім метод повертає накопичений результат.

Метод ParseFactor є найскладнішим методом у коді, оскільки він розбирає і обчислює множник, який може бути числом, змінною, викликом функції або виразом у круглих дужках. Спочатку метод перевіряє, чи поточна позиція у вхідному рядку не є кінцем, і якщо так, то генерує виключення. Якщо наступним символом у вхідному рядку є -, він зчитує наступний символ і рекурсивно викликає метод ParseFactor для розбору наступного множника, а потім повертає від'ємне значення результату. Якщо наступним символом є (, зчитується наступний символ і рекурсивно викликається метод ParseExpression для розбору виразу в дужках, а потім перевіряється, чи наступним символом є) і генерується виключення, якщо це не так. Якщо наступний символ є цифрою або крапкою, він зчитує наступні символи, поки не досягне нецифрового символу, а потім перетворює отриманий рядок у тип double за допомогою методу double.TryParse і повертає результат. Якщо наступний символ є літерою, він читає наступні символи, доки не досягне нелітерного символу, а потім перевіряє, чи є наступний символ (. Якщо це так, він зчитує наступні символи, розділені комами, поки не знайде (, і рекурсивно викликає метод ParseExpression для розбору кожного аргументу, а потім обчислює функцію на основі її імені за допомогою функцій бібліотеки Math. Якщо наступний символ не є (, програма припускає, що поточний рядок є іменем змінної, і шукає змінну у словнику змінних.

3.4. Тестування

Для проведення експерименту було випадково згенерували колекцію зі 100 виразів, які відрізнялися за довжиною та складністю. Вирази складалися з різної кількості операндів, діапазон яких охоплював від 10 до 10 000 операндів. Щоб запобігти спотворенню даних, буде рандомізовано порядок, у якому обчислюються вирази для кожного методу. Буде використовувати секундомір, щоб виміряти час, витрачений на кожен метод для оцінки кожного виразу.

У цьому експерименті час, витрачений для кожного методу на обчислення арифметичних виразів, буде записаний у мілісекундах. Після оцінки кожного виразу за допомогою кожного методу буде обчислено середній час, потрібний кожному методу для оцінки кожного виразу. Результати будуть представлені в таблиці, щоб краще наочно побачити відмінності між методами. Це також допоможе визначити, який метод є найбільш ефективним.

Для забезпечення точності та надійності даних буде проведено аналіз помилок. Будь-які викиди або аномалії в даних будуть виявлені та досліджені. Результати також порівнюватимуться з очікуваними результатами для забезпечення узгодженості.

Достовірність результатів також буде перевірено, щоб переконатися, що вони точно відображають ефективність методів. Статистична значущість результатів буде оцінена для визначення рівня впевненості у висновках, зроблених на основі даних. Крім того, будь-які фактори, що можуть вплинути на результати, перевірятимуться та контролюватимуться для підвищення точності та надійності результатів.

Загалом цей експеримент має на меті забезпечити комплексний аналіз ефективності різних методів обчислення арифметичних виразів. Збираючи та аналізуючи дані про час виконання, а також виконуючи аналіз помилок і перевірки достовірності, експеримент прагне отримати надійні та точні результати, які можуть стати основою для розробки більш ефективних алгоритмів для арифметичного оцінювання.

4 РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТУ

В результаті проведеного експерименту було порівняно ефективність трьох алгоритмів: алгоритм, створений на основі дерев виразів, алгоритм Reverse Polish Notation та алгоритм Recursive Descent.

Для порівняння алгоритмів було використано три вирази з різною складністю:

"10 * (10 + 1) ^ 2",

"31 * (10 + 1) ^ 5",

"23 * (4 + 1) ^ 2 * 3 + 4 * 2 / (1 - 5) ^ 2 ^ 3 / 3 + 4 ^ (1 / 2) + 1 + cos (1 / 2) + sin (1 / 2) - 2 + (tan (1 / 2)) + cos (0) * 2 + ln (tan (1 / 2)) + cos (0) + ln (exp (2)) ",

"10 * (10 + 1) ^ 2 * 3 + 4 * 2 / (1 - 5) ^ 2 ^ 3 / 3 + 4 ^ (1 / 2) + 1 + cos (1 / 2) + sin (1 / 2) - 2 + (tan (1 / 2)) + cos (0) * 2 + ln (tan (1 / 2)) + cos (0) + ln (exp (2)) ".

Проведемо інформаційну підготовку прийняття рішення з вибору необхідної кращого алгоритму (див. табл. 4.1):

- опишемо множину альтернатив:
  1) алгоритм на основі дерева рішень;
  2) алгоритм Reverse Polish Notation;
  3) алгоритм Recursive Descent.
- опишемо критерії вибору:
  1) гнучкість – можливість швидкого редагування арифметичного виразу, без потреби перетворення і заповнення проміжної структури даних;
  2) зручність використання – можливість пере використання проміжної структури даних для подальших процесів;
  3) продуктивність алгоритму при малій кількості даних – наскільки швидко відпрацьовує алгоритм з кроткими арифметичними виразами;

4) продуктивність алгоритму при великій кількості даних – наскільки швидко відпрацьовує алгоритм з великими, та довгими арифметичними виразами;

– опишемо шкали оцінок за критеріями:

1) гнучкість, зручність використання, продуктивність алгоритму при малій кількості даних, продуктивність алгоритму при великій кількості даних – кількісні показники які матимуть шкалу оцінювання від 1 до 3, 3 – це найвища оцінка, а 1 – найнижча.

Таблиця 4.1 – Дані спостереження

	Дерево рішень	RPN	Recursive Descent
Гнучкість	3	1	1
Зручність використання	3	2	2
Продуктивність алгоритму при великій кількості даних	2	2	2
Продуктивність алгоритму при малій кількості даних	1	2	3

За принципом Парето виключимо алгоритм RPN та матимемо наступне (див. табл. 4.2):

Таблиця 4.2 – Результативна таблиця

	Дерево рішень	Recursive Descent
Гнучкість	3	1
Зручність використання	3	2

Продовження таблиці

Продуктивність алгоритму при великій кількості даних	2	2
Продуктивність алгоритму при малій кількості даних	1	3

Тепер проведемо лінійну адитивну згортку (див. рис. 4.3):

Таблиця 4.3 – Згортка моделі

	Нормуючий множник	Дерево рішень	Recursive Descent
Гнучкість	0.25	3	1
Зручність використання	0.33	3	2
Продуктивність алгоритму при великій кількості даних	0.33	2	2
Продуктивність алгоритму при малій кількості даних	0.25	1	3
Результат згортки		2.65	2.32

Також було проведено тестування на роботу з великими виразами, що складаються з більше ніж 100 операндів, та було виявлено, що алгоритм Recursive Descent працює швидше з невеликими арифметичними виразами за два інших алгоритми, але при цьому має значну кількість помилок при роботі з великими виразами, що може вплинути на точність обчислень. Алгоритм Reverse Polish Notation працює також швидше з невеликими виразами, якщо порівнювати з алгоритмом який реалізує дерево рішень, але у останнього алгоритму є перевага

при роботі з довгими і складними виразами, а також з повторними виразами в яких були введені невеличкі зміни.

Отже, на основі проведених експериментів можна зробити висновок, що для роботи зі складними та довгими арифметичними виразами найбільш ефективним методом є алгоритм створений на основі дерева виразів. Він забезпечує можливість оновлення виразу без необхідності перетворення його наново та має надійність та точність обчислень. Алгоритм Reverse Polish Notation є швидшим за алгоритм на основі дерева виразів, але не має можливості оновлення виразу та може бути менш зрозумілим для людей, що працюють з програмою. Алгоритм Recursive Descent, хоча є найшвидшим, має високу кількість помилок при роботі з великими виразами та тому може бути менш практичним у застосуванні.

5 ПРОПОЗИЦІЇ ПОКРАЩЕННЯ АЛГОРИТМУ

Результати експерименту демонструють, що алгоритм, заснований на оцінці дерева виразів, забезпечує гнучке та ефективне рішення для обробки математичних виразів, особливо тих, які є великими та складними. Порівняно з іншими популярними алгоритмами, цей підхід вигідно відрізняється за часом виконання та можливостями обробки.

Однією з сильних сторін алгоритму є його опора на дерево виразів, що дозволяє ефективно обробляти вирази будь-якого розміру та складності. Ця особливість дозволяє алгоритму обробляти великі вирази, які можуть бути складними для інших алгоритмів, включаючи ті, що включають складні обчислення, функції або інші математичні операції.

Крім того, використання дерева виразів у алгоритмі забезпечує рівень гнучкості, якого немає в інших алгоритмах, що дозволяє адаптувати його до широкого спектру застосувань. Наприклад, дерево виразів можна легко розширити для підтримки нових математичних операцій або функцій, які можна додавати до дерева як нові вузли. Це робить алгоритм універсальним і адаптивним рішенням, яке можна використовувати в різних математичних додатках.

Крім того, переваги алгоритму в продуктивності над іншими алгоритмами роблять його привабливим рішенням для додатків, чутливих до часу, або тих, що вимагають ефективного використання ресурсів. Наприклад, підхід може добре підходити для використання в наукових обчисленнях, фінансовому аналізі або інших сферах, де обробка великих і складних виразів є поширеною вимогою. Експеримент показав, що алгоритм, заснований на оцінці дерева виразів, є потужним та ефективним інструментом для обробки математичних виразів будь-якого розміру та складності, і він пропонує значні переваги над іншими популярними алгоритмами з точки зору гнучкості та можливостей обробки.

Гнучкість алгоритму дозволяє легко налаштовувати і розширювати його для підтримки нових математичних операцій або функцій. Це робить його придатним для широкого спектру застосувань і варіантів використання в математичній галузі,

від наукових обчислень до фінансового аналізу тощо. Результати експерименту демонструють, що алгоритм, заснований на оцінці дерева виразів, є потужним та ефективним інструментом для обробки математичних виразів будь-якого розміру та складності, пропонуючи значні переваги над іншими популярними алгоритмами з точки зору гнучкості, можливостей обробки та використання пам'яті. Хоча все ще існують обмеження, які можна усунути, алгоритм являє собою багатообіцяючий підхід для оцінки постфіксних виразів, з потенціалом для подальших досліджень і розробок.

І оскільки ми вже торкнулися теми гнучкості, можна додати, що алгоритм є дуже модульним, що дозволяє легко модифікувати і налаштовувати його. Кожен вузол у дереві виразів представляє математичну операцію або операнд, і ці вузли можна легко додавати, видаляти або модифікувати за потреби для підтримки різних типів виразів. Така модульна структура також дозволяє легко інтегрувати алгоритм у більші програмні системи або фреймворки.

Хоча алгоритм, заснований на дереві виразів, добре показав себе в експериментах, є ще деякі області, де його можна вдосконалити. Одним з обмежень підходу є те, що він вимагає додаткового кроку побудови дерева виразів перед оцінкою, що може зайняти деякий час і пам'ять для дуже великих виразів. Іншим обмеженням є те, що алгоритм передбачає, що вхідний вираз має постфіксний запис, а це означає, що він не може бути використаний для обчислення виразів в інших форматах, таких як інфіксний або префіксний, без попереднього перетворення їх у постфіксний запис.

Одним з можливих шляхів покращення алгоритму є розробка гібридного підходу, який поєднує дерево виразів з іншими методами, такими як алгоритм маневреного двору або рекурсивний синтаксичний розбір, щоб забезпечити більшу гнучкість у форматах вхідних виразів. Інший підхід полягає в оптимізації побудови дерева виразів шляхом використання більш ефективних структур даних або алгоритмів.

Наступним з можливих покращень алгоритму парсингу арифметичних виразів, що використовує дерево рішень, є оптимізація обчислення часто

повторюваних виразів. При обчисленні складних виразів часто зустрічаються вирази, які повторюються більше одного разу. Замість того, щоб обчислювати ці вирази кожен раз, можна зберігати їх значення у хеш-таблиці.

Такий підхід дозволяє підвищити ефективність обчислення великих арифметичних виразів і скоротити час виконання програми. Для цього необхідно створити хеш-таблицю, в яку будуть зберігатися значення виразів та їх хеш-коди. Під час обчислення нового виразу, спочатку його хеш-код порівнюється з хеш-кодами вже збережених в таблиці виразів. Якщо хеш-коди співпадають, то значення виразу береться з таблиці, інакше обчислюється нове значення виразу і додається до таблиці з новим хеш-кодом.

Таким чином, оптимізація парсингу арифметичних виразів з використанням хеш-таблиці дозволяє зменшити час обчислення складних виразів і зберегти пам'ять, зберігаючи значення виразів лише один раз. При розробці такого підходу необхідно врахувати вимоги до швидкодії, розміру пам'яті та можливості реалізації відповідної хеш-функції.

Загалом, хоча алгоритм, заснований на дереві виразів, є багатообіцяючим підходом для оцінки постфіксних виразів, все ще є місце для подальших досліджень і розробок, щоб усунути деякі з його обмежень і оптимізувати його продуктивність.

ВИСНОВКИ

В процесі виконання дипломної роботи було досліджено алгоритми синтаксичного аналізу арифметичних виразів, проаналізовано можливі структури даних, які могли б використовуватися для зберігання операцій арифметичних виразів, а також було обрано найбільш підходящу структуру – дерево рішень.

Перевагами дерев рішень є:

- дуже зручне зберігання даних у ієрархічній системі;
- інтуїтивність дерев рішень;
- точність;
- гнучкість;
- можливість оновлення без перетворення усього дерева;
- швидкий процес навчання.

Класифікаційна модель, представлена у вигляді дерева рішень, є інтуїтивною та спрощує розуміння розв'язуваної задачі. Результат роботи алгоритмів конструювання дерев рішень, на відміну, наприклад, від нейронних мереж, що представляють собою "чорні ящики", легко інтерпретується користувачем. Ця властивість дерев рішень не тільки важлива при віднесенні до певного класу нового об'єкта, але і корисна при інтерпретації моделі класифікації в цілому. Дерево рішень дозволяє зрозуміти і пояснити, чому конкретний об'єкт відноситься до того або іншого класу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Використання дерев виразів. URL: https://jak.koshachek.com/articles/dereva-viraziv.html.
2. Наскрізний розбір тексту в математичні вирази. URL: https://aclanthology.org/D19-1536.pdf .
3. Дерева та графи. URL: https://www.cs.purdue.edu/homes/spa/courses/cs182/mod8.pdf.
4. Програмна та апаратна реалізація при генерації обчислювальних пристроїв. URL: https://www.math.spbu.ru/user/gran/sb1/yak.pdf.
5. Арифметичні аналізатори. URL: https://www.researchgate.net/figure/Parser-structure-for-simple-arithmetic-expressions-grammar_fig1_261223613.
6. K. Smelyakov, A. Chupryna, O. Bohomolov and N. Hunko, "The Neural Network Models Effectiveness for Face Detection and Face Recognition," 2021 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream), 2021, pp. 1-7, doi: 10.1109/eStream53087.2021.9431476.
7. K. Smelyakov, A. Chupryna, O. Bohomolov and I. Ruban, "The Neural Network Technologies Effectiveness for Face Detection," 2020 IEEE Third International Conference on Data Stream Mining & Processing (DSMP), 2020, pp. 201-205, doi: 10.1109/DSMP47368.2020.9204049.
8. K. Smelyakov, A. Datsenko, V. Skrypka and A. Akhundov, "The Efficiency of Images Reduction Algorithms with Small-Sized and Linear Details," 2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), 2019, pp. 745-750, doi: 10.1109/PICST47496.2019.9061250.
9. Expression Trees. URL: https://www.iue.tuwien.ac.at/phd/klima/node56.html.
10. Shunting yard algorithm. URL: https://en.wikipedia.org/wiki/Shunting_yard_algorithm.
11. Recursive Descent Parser. URL: https://www.tutorialspoint.com/what-is-recursive-descent-parser.