НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра цифрових технологій в енергетиці

"На правах рукопису"
УДК 004.032.2(043.3)

«До захисту допущено»
Завідувач кафедри
_____ Наталія АУШЕВА
"___"_____2022р.

Магістерська дисертація

зі спеціальності - 122 Комп'ютерні науки
за освітньо-професійною програмою магістерської підготовки - Комп'ютерний моніторинг та геометричне моделювання процесів і систем

на тему Обробка персистентних структур даних

Виконав: студент _2_курсу, групи_ТР-13мп__
Філіпенков Ілля Григорович _____
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник доц. Сидоренко Юлія Всеволодівна _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _доц. Міца Олександр Володимирович__ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ - 2022

Національний технічний університет України
“ Київський політехнічний інститут ім. Ігоря Сікорського”

Навчально-науковий інститут атомної та теплової енергетики
Кафедри цифрових технологій в енергетиці
Рівень вищої освіти другий, магістерський
За освітньою програмою "Комп’ютерний моніторинг та геометричне моделювання процесів і систем"
Спеціальності 122 Комп’ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Наталія АУШЕВА
(підпис)
«____» _____ 2022р.

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ ДИСЕРТАЦІЮ СТУДЕНТУ

Філіпенкову Іллі Григоровичу
(прізвище, ім’я, по батькові)

1. Тема дисертації Обробка персистентних структур даних

Науковий керівник Сидоренко Юлія Всеволодівна, к.т.н., доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від “7” листопада 2022 року № 4067-с
2. Строк подання студентом дисертації 9 грудня 2022 року
3. Вихідні дані до роботи Форма реалізації – бібліотека динамічного компонування, реалізована на базі платформи .NET 6 за допомогою мови програмування C# 10.0. Інтегроване середовище розробки – MS Visual Studio 2022.
4. Перелік питань, які потрібно розробити: Описати та реалізувати алгоритми перетворення структур даних на персистентні. Здійснити оцінку ресурсів часу та пам’яті, використовуваних описаними структурами. Створити програмне забезпечення, яке дозволило б використовувати персистентні структури даних у розробці програмних продуктів.

5. Орієнтований перелік ілюстративного матеріалу «Актуальність роботи», «Мета та поставлені задачі», «Об’єкт та предмет дослідження», «Практичне значення одержаних результатів», «Методи перетворення структур даних на персистентну», «Метод копіювання шляху», «Алгоритми реалізації персистентних версій поширених структур даних», «Оцінка ресурсів часу, використовуваних персистентними структурами даних», «Програмні засоби реалізації», «Діаграма класів», «Розробка стартап-проекту», «Висновки»

6. Орієнтований перелік публікацій _____
_____
_____
_____
_____
_____

8. Дата видачі завдання «____» _____ 2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання магістерської дисертації	Строки виконання етапів магістерської дисертації	Примітка
1	Отримати завдання	До 01.09.2022	
2	Практика	За графіком	
3	Збір інформації	До кінця практики	
4	Аналіз вимог завдання, розробка методів і засобів розв'язання поставленої задачі	До кінця практики	
5	Розробка та тестування програмного продукту	До кінця практики	
6	Виконання розділів дисертації (практична частина, загальні висновки, список джерел)	До 01.11.2022	
7	Написання основних розділів автореферату	До 25.10.2022	
8	Перевірка дисертації науковим керівником		
9	Подання в електронному вигляді роботи та анотації до неї на перевірку нормоконтролера та плагіат (UNICHECK)		
10	Надання документів на засідання кафедри	За день до засідання	
11	Передзахист магістерської дисертації та допуск до захисту дисертації	Згідно з планом кафедри	
12	Подання магістерської дисертації рецензенту. Отримання рецензії.	До подання пакету документів до ЕК	
13	Подання пакету документів за магістерською дисертацією та супровідних до неї документів до захисту в ЕК	За 5 днів до дати захисту за графіком	
14	Захист магістерської дисертації		

Студент _____ ___Ілля ФІЛІПЕНКОВ_____
(підпис) (прізвище та ініціали)

Науковий керівник _____ ___Юлія СИДОРЕНКО____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Актуальність роботи. Сучасні підходи до конструювання та розробки програмного забезпечення вимагають все ширшого застосування нових перспективних способів зберігання і обробки інформації. Якщо стоїть завдання мати доступ до усіх попередніх версій структури за той самий асимптотичний час, то персистентні структури даних є практично незамінними.

Персистентна структура даних – це структура, що зберігає свої попередні стани при кожній модифікації, таким чином, забезпечуючи можливість роботи з її станом в будь-який відрізок часу. Через те, що при кожній модифікації структура зберігає свою минулу версію, ми можемо працювати не тільки з поточними, але і з даними із попередніх станів структури, тобто її «минулим». Щоб розрізняти одну й ту саму структуру з різних періодів часу, кожен її стан ідентифікується особливим чином (в вигляді числа, вектора чисел чи хеша), даний ідентифікатор для персистентних структур називається її версією.

Мета роботи. Створення програмного забезпечення для обробки повністю персистентних структур даних, а саме стеку, черги та дерева відрізків, з використанням ефективних алгоритмів їх побудови.

Для досягнення поставленої мети в роботі визначені наступні **завдання**:

- описати та реалізувати алгоритми перетворення структур даних на персистентні;
- здійснити оцінку ресурсів часу та пам’яті, використовуваних описаними структурами;
- створити програмне забезпечення, яке дозволило б використовувати персистентні структури даних у розробці програмних продуктів.

Практичне значення одержаних результатів полягає у можливості використання персистентних структур даних з оптимальним часом та пам’яттю виконання для задач у розробці програмного забезпечення.

Структура і обсяг дипломної роботи. Магістерська дисертація складається зі вступу, п’яти розділів, висновків та одного додатку. Робота містить посилання на двадцять чотири джерела та двадцяти трьох ілюстрацій. Основна частина роботи викладена на шістдесяти восьми сторінках.

Ключові слова: *алгоритм, бінарне дерево, дерево відрізків, структура даних, персистентна структура даних.*

ABSTRACT

Relevance of work. Modern approaches to the design and development of software require an ever wider application of new promising ways of storing and processing information. If the task is to have access to all previous versions of the structure in the same asymptotic time, then persistent data structures are practically irreplaceable.

A persistent data structure is a structure that preserves its previous states with each modification, thus ensuring the possibility of working with its state at any time. Due to the fact that with each modification the structure preserves its previous version, we can work not only with current, but also with data from previous states of the structure, i.e. its "past". To distinguish the same structure from different periods of time, each of its states is identified in a special way (in the form of a number, a vector of numbers or a hash), this identifier for persistent structures is called its version.

The goal of the work. Develop software for processing fully persistent data structures, namely stack, queue and segment tree, using efficient algorithms for their construction.

To achieve the set goal, the following **tasks** are defined in the work:

- describe and implement algorithms for converting data structures into persistent ones;
- evaluate the time and memory resources used by the described structures;
- create software that would allow the use of persistent data structures in the development of software products.

The practical significance of the obtained results lies in the possibility of using persistent data structures with optimal execution time and memory for software development tasks.

Structure and scope of the graduate work. The master's graduate work consists of an introduction, five chapters, conclusion and one appendice. The work contains references to twenty-four sources and twenty-three illustrations. The main part of the work is laid out on sixty-eight pages.

Key words: *algorithm, binary tree, segment tree, data structure, persistent data structure.*

ЗМІСТ

ВСТУП 10

РОЗДІЛ 1 АНАЛІЗ МЕТОДІВ ПЕРЕТВОРЕННЯ СТРУКТУРИ ДАНИХ НА ПЕРСИСТЕНТНУ 12

1.1 Класифікація персистентних структур даних 12

1.2 Методи перетворення структури даних на персистентну 14

РОЗДІЛ 2 АНАЛІЗ ІСНУЮЧИХ ПРОГРАМНИХ ПРОДУКТІВ ДЛЯ ОБРОБКИ ПЕРСИСТЕНТНИХ СТРУКТУР ДАНИХ 22

2.1 Бібліотека Immutable Collections 22

2.2 Бібліотека C5 23

РОЗДІЛ 3 АЛГОРИТМИ РЕАЛІЗАЦІЇ ПЕРСИСТЕНТНИХ ВЕРСІЙ ПОШИРЕНИХ СТРУКТУР ДАНИХ 26

3.1 Задача програмного забезпечення обробки персистентних структур даних. 26

3.2 Персистентний стек 27

3.3 Персистентна черга 30

3.4 Персистентне дерево відрізків 37

РОЗДІЛ 4 ПРОГРАМНА РЕАЛІЗАЦІЯ ПЕРСИСТЕНТНИХ СТРУКТУР ДАНИХ 41

4.1 Засоби розробки програмного продукту 41

4.2 Опис реалізації системи 42

4.3 Оцінка використовуваних ресурсів часу та пам’яті 47

4.4 Інсталяція та системні вимоги 50

4.5 Деінсталяція програмної системи 52

РОЗДІЛ 5 РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ 55

5.1 Опис ідеї стартап-проекту 55
5.2 Технологічний аудит проекту 57
5.3 Аналіз ринкових можливостей запуску стартап-проекту 58
5.4 Розробка ринкової стратегії 68
5.5 Розробка маркетингової програми стартап-проекту 71
ВИСНОВКИ 76
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 78
ДОДАТОК А 80

ВСТУП

Розробка деяких прикладних інструментів та вирішення певного класу геометричних задач створили необхідність у використанні таких структур даних, які зберігають свої попередні стани, а також, за необхідності, мають можливість повертатися до них. Приклади використання цього підходу можна знайти у багатьох програмних додатках, таких як: системи управління версіями (git), сервіси геопозиціонування або редактори документів, що підтримують доступ до усіх попередніх версій. Якщо структура даних має таку функціональність, вона називається персистентною. Персистентні структури даних були вперше представлені у 1989 році [1].

Зазвичай, коли йде мова про персистентність у програмуванні, мається на увазі збереження даних у якесь сховище, наприклад, базу даних, таким чином, що ці дані можуть бути завантажені пізніше, коли додаток запустається знову. Однак, цей термін також може використовуватися до структур даних, наприклад, тих, що використовуються у функціональному програмуванні. У цьому контексті персистентна структура даних може зберігати усі версії при модифікуванні.

З іншого боку, для зберігання усіх версій структури даних необхідно виділяти додаткову пам’ять і витрачати час виконання на пошук необхідної версії. Дотримання цих вимог є доволі нетривіальною задачею і вирішується повторним використанням деяких даних.

При використанні сучасних підходів до моделювання та розробки програмного забезпечення все частіше виникає необхідність у використанні нових перспективних способів обробки і зберігання інформації. Застосування персистентних структур даних є практично незамінним, якщо виникає завдання доступу до усіх попередніх версій структури та можливості їх зміни. У разі необхідності використання таких структур у розробці програмного продукту, розробнику доведеться довгий час вивчати теоретичні основи персистентності, випробовувати різні методи реалізації та тестувати їх. Саме тому було поставлено

задачу створення програмного забезпечення обробки персистентних структур даних.

Необхідно створити програмне забезпечення, що надає можливість використовувати повністю персистентні структури даних у розробці.

Задачі, які мають вирішуватись програмним забезпеченням: використання повністю персистентних структур даних для загальних типів даних; оптимальний алгоритмічний час виконання та використання пам’яті; зручна інтеграція програмного забезпечення у інший програмний продукт.

Вхідна інформація для задачі побудови персистентної структури даних: тип структури даних, тип даних, що у ній зберігаються, кількість елементів та кількість версій, що будуть підтримуватися.

Вихідна інформація: об'єкт, що надає можливість виконувати всі дії, передбачені заданою структурою даних, звертатись та модифікувати попередні версії структурі даних.

Потенційними користувачами програмного забезпечення можуть бути розробники та інженери програмного забезпечення. Програмне забезпечення може бути впроваджено у системах контролю версіями або у технологічний комплекс, який розв'язує задачу локалізації точки з онлайн запитами.

1. АНАЛІЗ МЕТОДІВ ПЕРЕТВОРЕННЯ СТРУКТУРИ ДАНИХ НА ПЕРСИСТЕНТНУ

Персистентна структура даних – структура даних, яка зберігає свої попередні стани при кожній модифікації і забезпечує можливість роботи з будь-яким її станом у будь-який відрізок часу [3]. Оскільки при кожній модифікації зберігається минула версія, завжди можна працювати як з поточними, так і з даними із попередніх станів структури. Кожен із станів структури однозначно ідентифікується за номером, вектором або хешем. Така функціональність використовується для того, щоб розрізняти одну й ту саму структуру у різні періоди часу. Кожен зі станів називається версією, а ідентифікатор – номером версії.

Персистентні структури даних використовуються під час вирішення геометричних задач. Прикладом може бути задача локалізації точки з онлайн запитами.

У цьому розділі проаналізовано типи персистентності та методи перетворення структур даних на персистентні.

1.1 Класифікація персистентних структур даних

За функціональністю класифікують два типи персистентних структур даних: власне персистентні та ретроактивні. При внесенні змін до однієї з попередніх версій персистентної структури даних, початковий ланцюжок змін зберігається, а від модифікованої старої версії відгалужується новий. Усі версії, які слідують після відгалуження у старій гілці, залишаються незмінними. Таким чином, можна вносити зміни у минулі версії, не впливаючи на наступні.

Ретроактивними структурами даних передбачена інша функціональність. Зміни, внесені у минулі версії, вносяться також і у всі наступні. Отже, при

модифікації однієї із попередніх версій, стан структури у «теперішньому» також зміниться.

Виділяють такі види власне персистентних структур даних:

- частково персистентні;
- повністю персистентні;
- конфлюентні;
- функціональні.

Частково персистентні структури даних надають доступ до попередніх версій лише у режимі читання. Тобто, створені раніше версії не можуть стати родоначальниками нових. Змінюваною ж є лише остання версія. Граф змін у такому випадку буде виглядати як ланцюжок. Тому для ідентифікації версій доволі просто використовувати натуральні числа.

Повністю персистентні структури даних надають усю функціональність частково персистентних, а також можливість внесення змін у будь-яку з минулих версій структури. Граф версій матиме вигляд дерева, тобто при модифікації однієї з минулих версій створюється нова гілка. Існують структури даних, у яких часткова персистентність досягається тривіально, а повна є складною в реалізації. У роботі увага сфокусована саме на повністю персистентні структури даних.

Конфлюентні структури даних надають функціональність для об’єднання двох версій структури даних в одну.

У функціональних структурах даних неможливо зробити присвоювання, яке перезаписує дані, отже, повністю персистентні за означенням. Зміна значення після першого присвоєння неможлива.

Функціональна структура даних завжди є конфлюентною, конфлюнтна – завжди повністю персистентна, повністю персистентна – частково персистентною. Однак, не всі конфлюентні є функціональними [4].

1.2 Методи перетворення структури даних на персистентну

Класифікують три основні методи для того, щоб реалізувати персистентну структуру даних:

- повне копіювання;
- копіювання шляху;
- метод «товстих» вузлів.

Найбільш простим є метод повного копіювання. Він полягає в тому, що кожен раз, коли у структуру даних вноситься зміна, уся структура повністю копіюється, а у таблицю версій додається вказівник на неї. Цей метод є найбільш тривіальним і водночас найбільш затратним, як у плані споживання потужності процесора, так і пам’яті. Але оскільки більшість елементів у структурі даних не зазнають змін, такі витрати є абсолютно не виправданими. Елементи, маючи ті самі значення, дублюються і зберігаються у пам’яті.

Для зменшення кількості копійованих елементів використовується метод копіювання шляху. Метод полягає у копіюванні лише тих елементів, з яких модифікований є досяжним, тобто, лише тих, які є сильно зв’язані з ним. Таким чином, при створенні нової версії бінарного дерева, будуть копіюватися лише батьківські елементи до модифікованого. Отже, порівнюючи цей метод з повним копіюванням, отримуємо економію значної кількості пам’яті. У той же час, все ще залишаються не змінені елементи, які дублюються.

Для зведення витрат пам’яті до мінімуму використовується метод «товстих» вузлів. Однак, його можна застосувати не для усіх структур даних, а лише для тих, які представляються у вигляді «машини вказівників». Для цього структура даних має задовольняти такі умови:

- до складу структури даних входять тільки вказівники і вузли з даними;
- всі вузли мають константну кількість полів, тобто мають фіксований розмір;
- кожен вказівник має зворотній вказівник.

Таким чином, «машина вказівників» може бути представлена таким чином, що у кожному її вузлі буде зберігатися лог змін. Тоді при модифікуванні вузла, у його лог змін вноситиметься новий запис, але поле зі значенням самого «товстого» вузла змінюватись не буде. Записи у логу змін являють собою пари значень – номер версії, у якій зроблено модифікацію і нове значення. Коли викликається конкретна версія структури, у лозі змін шукається відповідна версія і значення з цього запису повертається.

Серед недоліків методу «товстих» вузлів варто звернути увагу на випадок, коли сворюється дуже багато версій. У разі, якщо виникає переповнення логу вузла, треба створити новий вузол. Значення з максимальної версії попереднього вузла буде початковим для нового. У такій ситуації, можливо, доведеться обійти всю структуру для оновлення вказівників. При такому алгоритмі більшість операцій над вузлом все одно відбуватимуться за O(1). Таким чином, будь-які операції над «товстими» вузлами матимуть амортизовану складність O(1).

Два останніх методи варто розглянути детальніше.

Метод копіювання шляху

Для легшого пояснення варто розглянути цей метод на збалансованому дереві пошуку. Якщо h – висота дерева, то усі операції з деревом можна зробити за O(h), висота дерева складає O(log(n)), де n – кількість вершин. Нехай стоїть завдання додати черговий елемент до збалансованого дерева, не втрачаючи старе дерево. Тоді потрібно не просто додати нового сина у якийсь вузол, а зробити копію батьківського вузла і до нього додати нового сина. Так само доведеться копіювати усі вузли та вказівники, починаючи з кореня, з яких є досяжним батьківський. При використанні такого алгоритму незміненими залишаться ті вершини, з яких модифікований вузол є недосяжним. Копійований шлях завжди матиме довжину рівну висоті дерева, тобто, O(log(n)), де n – кількість вершин. Таким чином, після завершення операції, маємо доступ до обох версій дерева.

При балансуванні дерева пошуку доведеться піднімати вверх вершину і копіювати усі, які беруть участь у обертанні, оскільки у них змінюються посилання

на дітей. Таких вершин завжди не більше трьох. Таким чином, асимптотика операції залишиться O(log(*n*)).

Після завершення балансування потрібно скопіювати усі вершини на шляху до кореня.

Метод копіювання шляху дозволяє зекономити використовувану пам'ять у структурах даних, які мають відносно небагато прив'язаних елементів до модифікованого (порядка логарифма від загальної кількості). Для стеку або двійкових дерев застосувати цей метод доволі просто, на відміну від черги. Оскільки черга має усі елементи зв'язані вказівниками із хвостом, операція додавання виявиться дуже затратною – доведеться копіювати усю структуру. Якщо у структурі даних є посилання на предка, то метод копіювання шляху також не підходить через затратне по пам'яті рішення.

Метод «товстих» вузлів

Розглянемо ту саму задачу, що і в методі копіювання шляху, де потрібно змінити певний елемент у структурі даних, не втрачаючи стару версію вузла. На рисунку 1.1 структура *X, V₁* має поле *b* = 3, а у *X, V₂* воно має бути рівне 4. Для початку, знехтуємо витратами часу. Метод «товстих» вузлів пропонує зберігати усі версії вузла в одному комбінованому «товстому» вузлі.

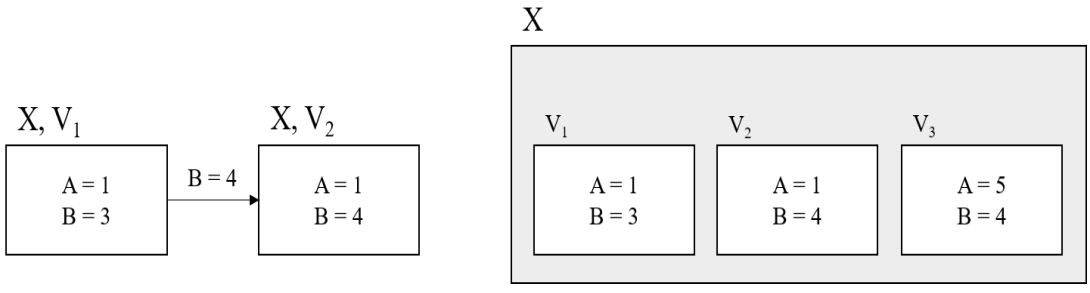

Рисунок 1.1 – Зміна *b* = 4

Отже, у «товстому» вузлі *X* будемо зберігати усі його версії. На рисунку 1.1 у ньому знаходяться V_1 і V_2, у яких *b* = 3 і *b* = 4, а також V_3, у якому модифіковане поле *a*.

У «товстому» вузлі зміниться алгоритм доступу до певної версії вузла. Сам вузол тепер складається з багатьох версій, тому доведеться здійснити пошук по списку версій. Шуканою версією буде та, номер якої менший або рівний номеру в запиті. Після здійснення пошуку вузол з цієї версії доступний для читання або зміни, як і звичайний вузол. Швидкий пошук по версіям можна реалізувати, зберігаючи їх у вигляді дерева. Таким чином, будь-яка операція над структурою даних матиме гіршу асимптотику, а саме – сповільнену на логарифм від кількості версій.

«Товстий» вузол можна реалізувати за іншим алгоритмом. Зберігатиметься не сам вузол, а зміна, яка відбулась. Тобто, лог змін міститиме пари значень: ідентифікатор версії і сама зміна. Лог вершини можна організувати по-різному. Найчастіше кожне поле вершини матиме свій лог змін. Модифікація вузла означатиме, що у лог додається запис про номер версії та оновлене значення поля. Для ефективного пошуку за логарифм версій записи у логу також зберігаються у вигляді дерева.

Дві варіації методу «товстих» вузлів мають не багато відмінностей. Асимптотично операції над структурою даних будуть відбуватися за однаковий час: $O(\log(t))$, де t – кількість змін структури даних; пам’яті необхідно $O(n + t)$, де n – число вершин у структурі даних [3].

Комбінований метод для часткової персистентності

Нехай є структура даних, у кожного вузла якої кількість вказівників на цей вузол не більша за деяку константу N. При клонуванні вузла важливо знати, звідки на цей вузол йдуть вказівники, щоб потім їх переставити. Тому необхідно у кожному вузлі зберігати зворотні посилання ті вузли, які посилаються на вузол, що клонується. Всі вузли будуть зберігатися у вигляді «товстих» вузлів, в яких міститься початкова версія цього вузла та список внесених до нього змін завдовжки не більше $2N$.

Нехай потрібно внести зміну до структури даних у вузол X. Якщо є місце у списку змін, туди просто вноситься зміна: записується номер версії, з якої починається ця зміна, до якого поля вузла вноситься зміна та яка саме. Якщо лог

змін заповнений, то вузол *X* клонується: береться стартова версія вузла, в ній робляться всі зміни, записані в логу змін, додається остання зміна та створюється версія з порожнім списком змін. Після цього необхідно пройти за зворотними посиланнями від *X* і в лог змін кожного вузла, що посилається на *X*, додати зміну вказівника починаючи з цієї версії структури даних з *X* на *X'*.

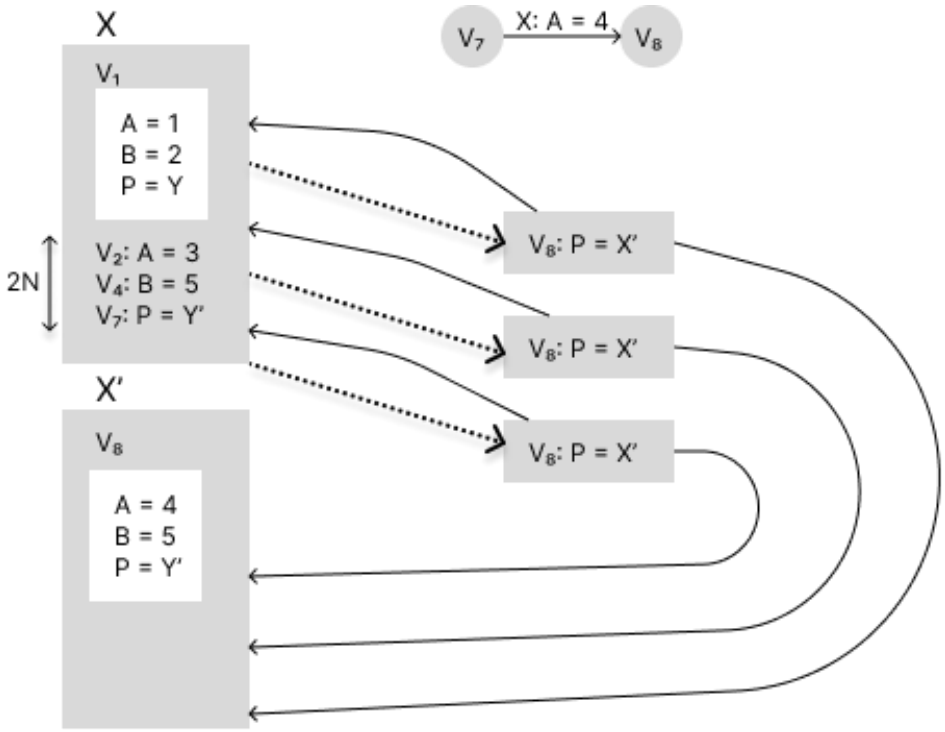

Рисунок 1.2 – Часткова персистентність

Для оцінки часу роботи цього алгоритму, нехай функція потенціалу дорівнюватиме сумарному розміру всіх списків змін в останній версії. Подивимося, як змінюється сумарний розмір списків змін, коли здійснюється одна зміна. Якщо лог змін був не повний, туди додається один елемент, потенціал збільшується на одиницю. Якщо лог змін був повний, то потенціал зменшується на його розмір, оскільки клонується вузол із порожнім списком змін. Після цього слідує обхід за зворотними посиланнями (*N* посилань) і додається *N* вузлів за одним значенням. Таким чином, амортизований час роботи буде O(1).

Комбінований метод для повної персистентності

Для повністю персистентних структур даних застосувати описаний вище метод перетворення не вийде, оскільки історія створення версій не лінійна і не можна відсортувати зміни за версіями, як у частково персистентних структурах даних. Нехай історія змін версій зберігається у вигляді дерева. Здійснивши обхід дерева углиб, записуватимемо послідовність входу та виходу з кожної версії та сформуємо з неї список. Коли після якоїсь версії (на рисунку 1.3 це версія 6) додається нова версія структури даних (на рисунку 1.3 це версія 8), до списку вставляються два елементи (на рисунку 1.4 це +8 та −8) після входу, але до виходу з тієї версії, коли відбулася зміна (тобто між елементами +6 та −6). Перший елемент вносить зміну, а другий повертатиме назад значення попередньої версії. Таким чином кожна операція розбивається на дві: перша робить зміну, а друга його відкочує.

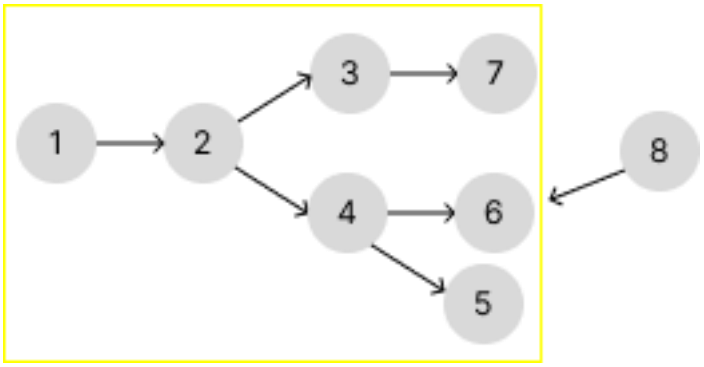

Рисунок 1.3 – Граф версій, до якого додається 8 версія

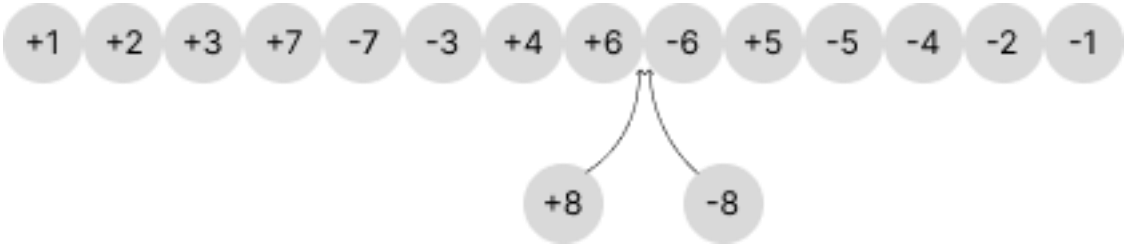

Рисунок 1.4 – Додавання нових елементів до списку

Описаний в попередньому пункті метод перетворення структур даних у повністю персистентні реалізовується за допомогою списку, який підтримує операції *insertAfter(p, q)* (вставка *q* після *p*) та *order(p, q)* (має відповідати на запити виду «*p* лежить у цьому списку до *q*»). Це список із підтримкою запиту про порядок, який обидві операції робить за O(1).

У лозі змін «товстого» вузла тепер додаються дві події: одна вказує на зміну, що сталася у відповідній версії, а інша на його скасування. Події додаватимуться в лог змін не за номерами версій, а за їх порядком у списку версій.

Коли є запит до якоїсь версії, потрібно знайти у списку версій таку, після входу до якої, але до виходу з якої лежить версія запиту, а серед таких максимальну. Наприклад, якщо надходить запит до версії 6 на рисунку 1.4, у списку версій вона лежить після входу, але до виходу з версій 1, 2 і 4. Необхідно знайти найбільшу з них. Описаний вище список дозволяє робити це за O(1) за допомогою операції *order(p, q)*. На рисунку 1.4 це версія 4. Оскільки лог змін кожного вузла має константний розмір, пошук потрібної версії у ньому відбувається за O(1).

Якоїсь миті лог змін «товстого» вузла переповниться. Тоді цей вузол клонується, а нижня половина змін переноситься до логу змін клонованого вузла. Перша половина змін застосовується до вихідної версії вузла і зберігається як вихідна у клонованому вузлі.

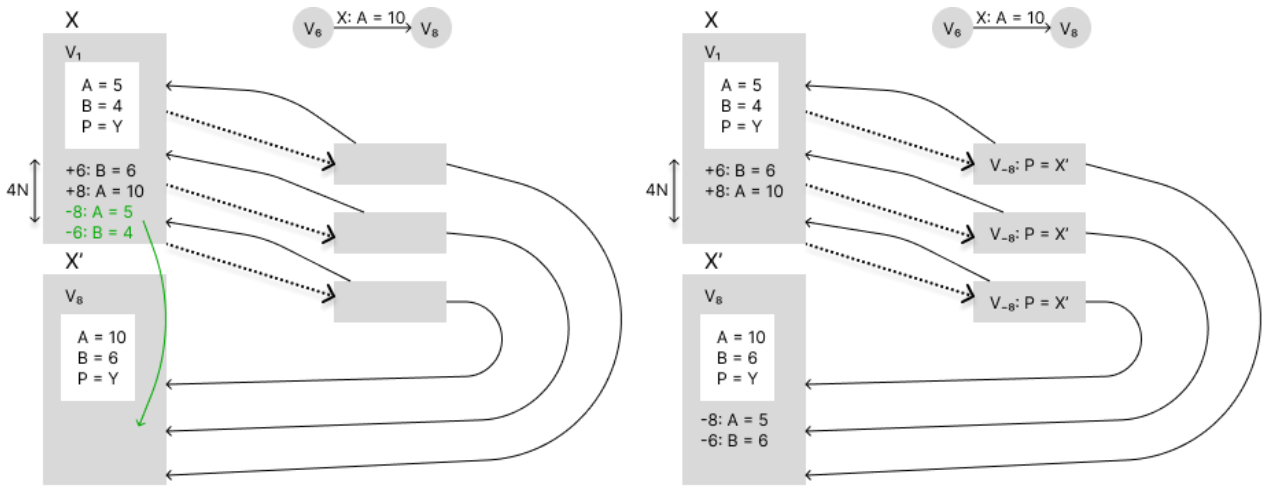

Рисунок 1.5 – Клонування «товстого» вузла

Після клонування є два вузли: перший відповідає за відрізок версій до операції останньої зміни, а другий — після неї. Подальший порядок дій аналогічний до того, що використовувався у загальному методі побудови частково персистентних структур даних.

Для оцінки амортизованого часу роботи цього алгоритму, нехай функція потенціалу дорівнюватиме кількості повних вузлів. Коли вузол роздвоюється, функція потенціалу зменшується на одиницю, потім переставляється N посилань, потенціал збільшується на N, отже, амортизаційний час роботи — O(1).

Висновки до розділу 1

1. Незважаючи на те, що часткова персистентність зазвичай досягається доволі просто для будь-яких структур даних, повна персистентність може бути складною в реалізації і досягатися зовсім іншим алгоритмом.
2. Метод копіювання шляху дозволяє зберегти значну кількість пам’яті в порівнянні з повним копіюванням, оскільки копіює лише зв’язані з модифікованим вузли. Однак, часто цей підхід не є найоптимальнішим.
3. Метод «товстих вузлів» зводить витрати пам’яті до мінімуму і є застосовним для усіх структур, які можна представити у моделі «машини вказівників». Недоліки цього методу виявляються, коли створюється багато версій і виникає необхідність обходу всієї структури.
4. Якщо задача з перетворення структури даних на персистентну не вирішується за допомогою вищезгаданих методів, але структура даних може бути представлена у вигляді «машини вказівників», то використовується комбінований метод, який є складнішим у реалізації.

2. АНАЛІЗ ІСНУЮЧИХ ПРОГРАМНИХ ПРОДУКТІВ ДЛЯ ОБРОБКИ ПЕРСИСТЕНТНИХ СТРУКТУР ДАНИХ

Існують реалізації обробки персистентних структур даних на більшості сучасних мов програмування. Але для платформи .NET та мови C# їх є не так багато.

Бібліотека Immutable Collections [5] надає функціональність для роботи з персистентними та іммутабельними структурами даних. Персистентність реалізована за методом копіювання шляху.

Бібліотека C5 [6] надає широкий спектр класичних структур даних, багату функціональність, найкращу можливу асимптотичну часову складність, задокументовану продуктивність і ретельно перевірену реалізацію.

На даному етапі персистентні структури даних не набули широкого розповсюдження, особливо на мові C#. Але вони мають перспективу, оскільки забезпечують ефективне використання пам’яті при тому самому асимптотичному часі, що і звичайні.

2.1 Бібліотека Immutable Collections

Бібліотека була створена американським розробником Леслі Сенфордом у 2005 році. У ній реалізовані іммутабельні версії стеку, масиву, відсортованого списку і списку з вільним доступом, а також персистентне АВЛ-дерево. Ці класи колекцій поступаються стандарним з System.Collections у швидкодії, але вони, насправді, і не призначені для цього.

Бібліотека розроблена на .NET Framework 1.1. Розповсюджується за ліцензією MIT. Тобто, при підключенні до проекту, всі інші файли проекту можуть бути розповсюджені за будь-якою іншою ліцензією.

Для використання бібліотеки необхідно завантажити вихідний код та зробити збірку проекту вручну у режимі Release. Потім можна підключати бібліотеку до свого рішення, додавши посилання на збірку у необхідному проекті.

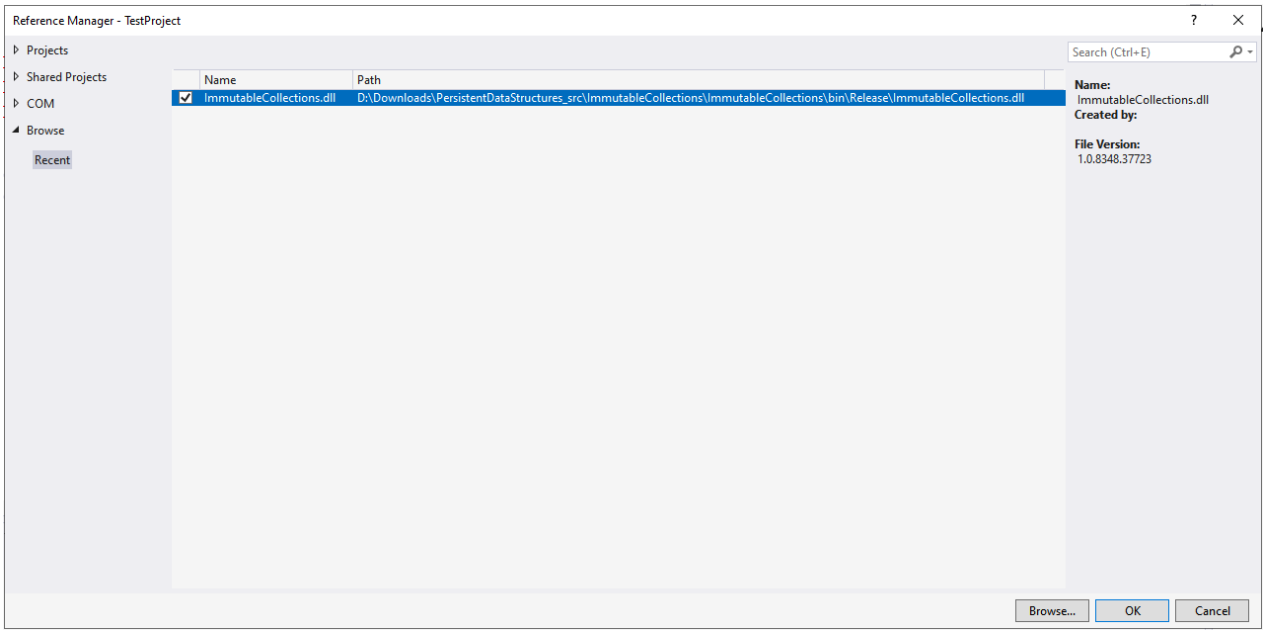

Рисунок 2.1 – Додавання посилання на бібліотеку Immutable Collections

Недоліками бібліотеки Immutable Collections є те, що у ній реалізовано мало структур даних. До того ж, реалізовані іммутабельні версії, які потребують додаткового коду, щоб перетворити їх на персистентні з контролем над версіями. Також бібліотека має лише свою першу версію та не вдосконалюється і не підтримується.

2.2 Бібліотека C5

Бібліотека C5 — це набір загальних класів колекції (або класів-контейнерів) для мови програмування C# та інших мов із підтримкою узагальнень на пізніших версіях платформи CLI, реалізованих у Microsoft .NET Framework 4.6.1+, .NET Core 2.0+. , .NET 5.0+ і Mono. Бібліотека була розроблена Нільсом Кохольмом, Пітером Сестофтом і Расмусом Лістроем. Розповсюджується за ліцензією MIT.

C5 є бібліотекою узагальнених колекцій для мови програмування C# та Common Language Infrastructure (CLI), функціональність, ефективність і якість якої відповідає або перевершує те, що доступно для аналогічних сучасних платформ програмування. На дизайн вплинули колекції бібліотек для Java і SmallTalk і їх опублікована критика. Однак вона містить функціональність і регулярність дизайну, які значно перевищують стандартні бібліотеки для цих мов. На момент написання розробка та реалізація бібліотеки завершені, а також написані та систематично застосовані обширні модульні тести.

З персистентних структур даних у бібліотеці реалізоване лише АВЛ-дерево і лише частково персистентна версія. Реалізоване воно за методом копіювання шляху.

Бібліотека опублікована на сайті nuget.org. Встановити можна за допомогою Package Manager, .NET CLI, PackageReference або Paket CLI. Для збірки проекту модульного тестування потрібен NUnit. Якщо у Visual Studio встановлено NuGet, він повинен автоматично додати посилання.

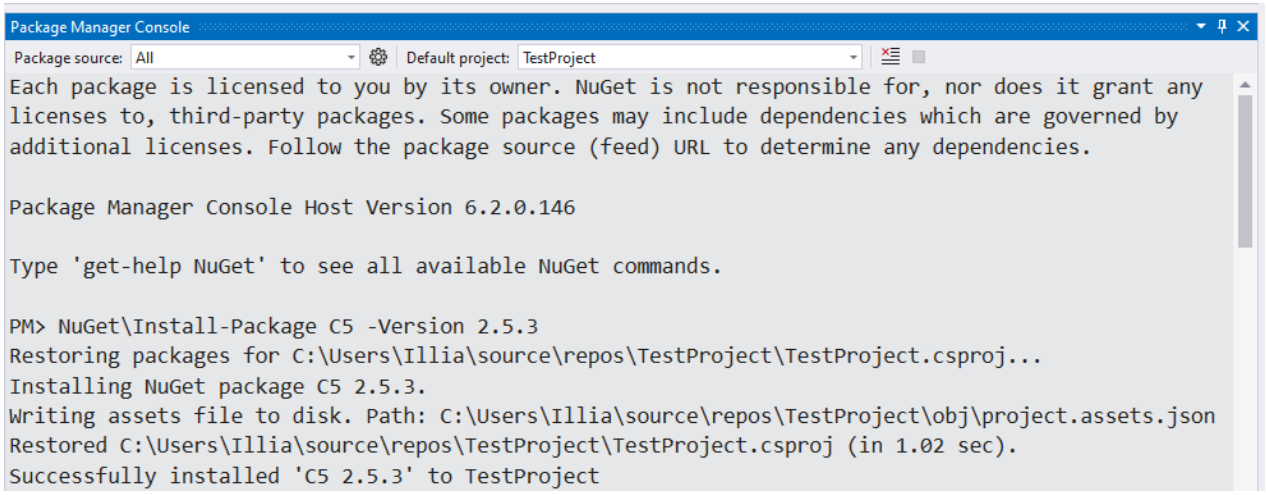

Рисунок 2.2 – Встановлення C5 через Package Manager

Незважаючи на те, що C5 має реалізації колекцій для багатьох структур даних, серед персистентних є лише реалізація АВЛ-дерева. Також серед недоліків варто зазначити, що ця реалізація лише частково персистентна.

Висновки до розділу 2

1. Враховуючи особливості поставленої задачі, для визначення можливостей вирішення задачі були проаналізовані існуючі програмні продукти, а саме:
    - бібліотека ImmutableCollections для роботи з персистентними та іммутабельними структурами даних;
    - бібліотека узагальнених колекцій для мови програмування C# та Common Language Infrastructure (CLI) C5.
2. Проаналізовано недоліки цих бібліотек та обґрунтована необхідність розробки нової бібліотеки.

3. АЛГОРИТМИ РЕАЛІЗАЦІЇ ПЕРСИСТЕНТНИХ ВЕРСІЙ ПОШИРЕНИХ СТРУКТУР ДАНИХ

У статті «Making Data Structures Persistent» [1] були розглянуті підходи та методи для створення загальної теоретичної основи персистентних структур даних. Це дозволило розвинути алгоритми реалізації та отримати декілька варіантів перетворення ефемерних структур даних на персистентні.

Наявність автоматичного збирача сміття у платформі .NET надає можливість створення нових бібліотек з реалізацією частково та повністю персистнентних структур даних, без проблем з витоком пам’яті.

3.1 Задача програмного забезпечення обробки персистентних структур даних

При використанні сучасних підходів до моделювання та розробки програмного забезпечення все частіше виникає необхідність у використанні нових перспективних способів обробки і зберігання інформації. Застосування персистентних структур даних є практично незамінним, якщо виникає завдання доступу до усіх попередніх версій структури та можливості їх зміни. У разі необхідності використання таких структур у розробці програмного продукту, розробнику доведеться довгий час вивчати теоретичні основи персистентності, випробовувати різні методи реалізації та тестувати їх. Саме тому було поставлено задачу створення програмного забезпечення обробки персистентних структур даних.

Для полегшення інтеграції персистентних структур даних у прграмні продукти, було створено програмне забезпечення, яке їх реалізовує та надає простий доступ до їхньої зміни.

Програмне забезпечення буде вирішувати такі задачі: використання повністю персистентних структур даних для загальних типів даних; оптимальний алгоритмічний час виконання та використання пам’яті; зручна інтеграція програмного забезпечення у інший програмний продукт.

Для того, щоб побудувати персистентну структуру даних за допомогою розробленого програмного забезпечення, необхідні будуть такі дані: тип структури даних, тип даних, що у ній зберігаються, кількість елементів та кількість версій, що будуть підтримуватися.

У результаті отримується об'єкт, що надає можливість виконувати всі дії, передбачені заданою структурою даних, звертатись та модифікувати попередні версії структурі даних.

Потенційними користувачами програмного забезпечення можуть бути розробники та інженери програмного забезпечення. Програмне забезпечення може бути впроваджено у системах контролю версіями або у технологічний комплекс, який розв'язує задачу локалізації точки з онлайн запитами.

3.2 Персистентний стек

Стек – це послідовний список, що має змінювану довжину та реалізовує додавання та видалення нових елементів за принципом «останнім прийшов – першим пішов» (англ. LIFO – Last In – First Out). Базові операції над стеком – додавання та виключення елемента з вершини стека. У якості додаткових операцій часто реалізовують очищення стека та визначення поточного числа елементів у ньому [5].

3.2.1 Опис вимог

Нехай існує пустий стек під номером 0. Змінна n відповідатиме за кількість стеків і на даному етапі рівна 1. Потрібно реалізувати базові операції над стеком:

- push(i, x) – додавання елементу зі значенням x в стек номер i. У результаті отримується стек під номером $n + 1$.

- pop(i) – вилучення останнього елементу зі стека номер i і повернути його значення. У результаті отримується стек під номером $n + 1$.

Таким чином, кожна операція має створювати новий стек, залишаючи доступ до старого.

3.2.2 Розв’язок

Розв'язок за допомогою методу повного копіювання полягає у симуляції описаного процесу, тобто копіюванні стека після кожної операції. Очевидно, що такий підхід є неефективним. Асимптотична складність цього алгоритму складе $O(n \cdot n)$ пам’яті та $O(n)$ часу.

Оптимізуємо рішення за допомогою методу копіювання шляху. Стек можна представити у вигляді графа, де елемент стека буде вершиною графа. У якості ребер графа візьмемо посилання на попередній елемент стека. Таким чином, усі вершини, окрім хвоста будуть зв’язані з попередньою. На рисунку 3.1 зображено граф, який представляє стек з п’яти елементів.

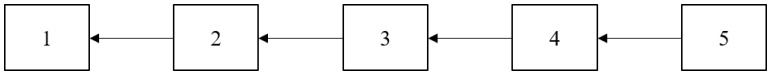

Рисунок 3.1 – Приклад стеку

Виходячи з даної побудови можна зробити висновок, що стек задається першим елементом, тобто, головою, з якої легко може бути відновленим. Отже, замість зберігання копій усіх елементів, можна зберігати тільки n перших елементів. Базові операції будуть реалізовуватися наступним чином:

- push(x, i) – новий елемент зі значенням x додається у голову стека. Новий елемент має посилання на елемент під номером i, який є головою версії під цим номером.
- pop(i) – повертає значення, елементу з номером i і копіює попередній елемент.

Оскільки жодна з операцій не робить копію додаткових елементів, то витрати пам’яті для підтримання черги буде $O(n)$, де n – кількість операцій додавання

елемента. Асимптотичний час операцій додавання та вилучення елемента складає O(1).

3.2.3 Приклад

Розглянемо описаний алгоритм на прикладі [6]. За умовою, існує пустий стек. Зберігатися він буде у вигляді голови стека із поміткою, що він пустий (рисунок 3.2).

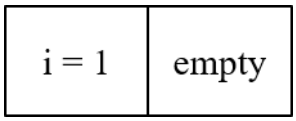

Рисунок 3.2 – Пустий стек

Здійснюємо операцію push(1, 3). У результаті отримуємо нову вершину зі значенням 3 та посиланням на першу (рисунок 3.3).

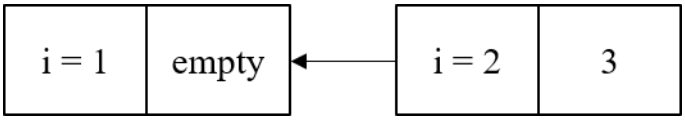

Рисунок 3.3 – Стек з одного елемента

Після операцій push(2, 7) і push(1, 4) отримаємо граф, зображений на рисунку 3.4.

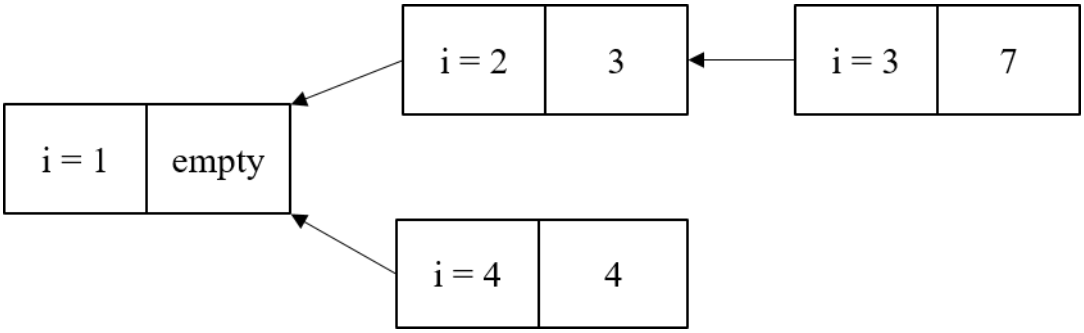

Рисунок 3.4 – Додано 3 елементи

Оскільки зберігаються голови усіх стеків, будь-який з чотирьох уже створених стеків відновлюється завдяки вказівникам. На рисунку 3.5 зображений граф після виконання операцій pop(2) і pop(3).

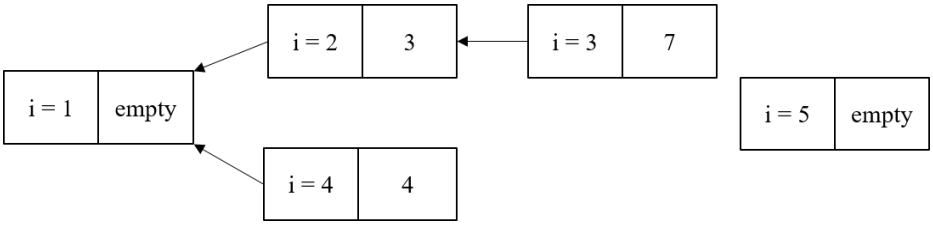

Рисунок 3.5 – Новий пустий стек

Після операції pop(3) повертається значення 7 і копіюється друга вершина, яка є головою шостого стека (рисунок 3.6).

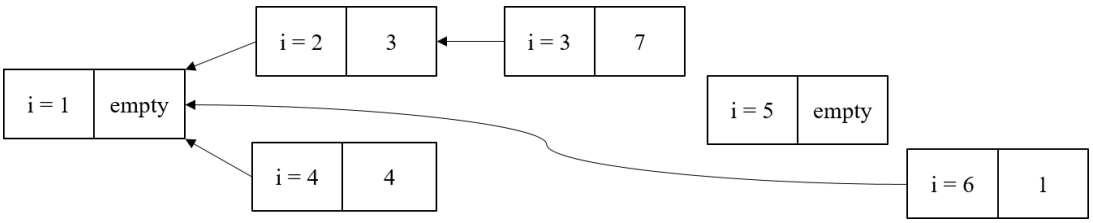

Рисунок 3.6 – Отримуємо шостий стек

3.3 Персистентна черга

Черга – це послідовний список, що має змінювану довжину та реалізовує додавання та видалення нових елементів за принципом «першим прийшов – першим пішов» (англ. FIFO – First In – First Out). Базові операції над чергою – додавання елементу в хвіст черги та виключення елемента з голови черги. У якості додаткових операцій часто реалізовують очищення черги, неруйнуюче читання та визначення поточного числа елементів у ній [5].

3.3.1 Опис вимог

Нехай існує пуста черга під номером 0. Змінна *n* відповідатиме за кількість черг і на даному етапі рівна 1. Потрібно реалізувати базові операції над чергою:

- push(i, x) – додавання елементу зі значенням x в голову черги номер i. У результаті отримується черга під номером n + 1.
- pop(i) – вилучення останнього елементу із черги номер i і повернути його значення. У результаті отримується черга під номером n + 1.

Таким чином, кожна операція має створювати нову чергу, залишаючи доступ до старої.

3.3.2 Розв'язок на двох стеках

Звичайна черга доволі просто реалізовується на двох стеках. Один стек віповідає за вилучення елементів, другий – за додавання. Коли у першому стеку закінчилися елементи, усі елементи з другого стеку переносяться в перший. Зробивши обидва стеки персистентними, отримуємо персистентну чергу. Однак, це рішення не є оптимальним навіть попри асимптотичну оцінку складномті операцій у O(1), оскільки оцінка є амортизованою. При перетворенні структури даних на персистентну, амортизовані оцінки не зберігаються. Тому необхідно розлянути інші варіанти реалізації, де використовується більша кількість стеків.

3.3.3 Розв'язок на п'яти стеках

Одним із мінусів реалізації на двох стеках є те, що в гіршому випадку витрачається O(n) часу на операцію. Якщо розподілити час, необхідний на переміщення елементів з одного стека в інший, за операціями, то можна отримати чергу без гірших випадків з O(1) істинного часу операцію.

Для реалізації персистентної черги на п'яти стеках, спочатку необхідно реалізувати звичайну чергу на шести, а потім перетворити її на персистентну.

Спочатку алгоритм схожий на випадок з двома стеками. Нехай існує стек L для операцій push і стек R для операцій pop. На момент спустошення стека R необхідно встигнути отримати стек R', що містить поточні елементи стека L у правильному для отримання порядку. Перекопіювання (recopy mode) почнеться, коли з'явиться небезпека того, що за кількість $R.size$, що залишилися, операцій pop зі стеком R неможливо перекопіювати стек L в новий стек R'. Вочевидь, що це ситуація $L.size > R.size$, нехай такий стан відображає спеціальна змінна логічного типу *recopy*.

Зрозуміло, що під час перекопіювання можуть надійти операції push, а стек *L* у цей час втратить свою структуру, скласти елементи туди вже не можна, отже, потрібно завести ще один стек *L′*, у який будуть складатися нові елементи. Після закінчення перекопіювання стеки *L*, *L′* і *R*, *R′* змінюються ролями, і на перший погляд, персистентна черга реалізована.

Однак, якщо реалізувати цей алгоритм, виявляється така проблема: старий стек *R* може і не спустошитися за цей час, тобто отримано два стеки з вихідними даними, а значить, можливий випадок (наприклад, якщо всі операції, що надходять - push), коли при наступному перекопіюванні не буде вільного стеку для копіювання туди елементів *L*. Для подолання цієї проблеми, примусово будуть вилучатися всі елементи зі стеку *R* у допоміжний стек *S*, потім елементи копіюватимуться зі стеку *L* в *R*, а потім елементи зі стека *S* будуть копіюватися в *R*. Легко показати, що наведений алгоритм отримує на виході в *R* всі елементи стеків *L*, *R* в правильному порядку.

Але цього ще замало. Якщо ми примусово виймаємо елементи зі стека *R*, виникають такі проблеми:

- Що повернути під час операції pop? Для цього треба завести стек *Rc* - копію стека *R*, з якого й будуть видобуватися необхідні елементи.
- Як підтримувати правильність такої копії? Оскільки цей стек потрібен тільки для перекопіювання, а під час нього він зайнятий, потрібна запасна копія *Rc* для копіювання всіх елементів, які копіюються в *R*, а після закінчення перекопіювання стеки *Rc*, *Rc′* міняються ролями, так само, як і стеки *L*, *L′*.
- Як врахувати, що під час перекопіювання частину елементів було вилучено з *Rc*? Для цього потрібна спеціальна змінна *toCopy*, яка показує, скільки коректних елементів знаходиться в стеку *S*, і зменшується при кожному вилученні з *S* або операції pop. Всі некоректні елементи будуть наростати з дна стека, тому ніколи не виникне ситуації, коли вилучається некоректний елемент, якщо *toCopy* > 0. Якщо під час операції pop *toCopy* = 0, це означає, що тепер у стеку *R* знаходиться весь правий шматок черги, тому доведеться витягти елемент з нього.

Тепер може виникнути проблема з непустим *Rc* після завершення перекопіювання. Покажемо, що він завжди може бути спустошеним, якщо буде використовуватися додаткове вилучення з нього при кожній операції у звичайному режимі, для цього повністю проаналізуємо алгоритм.

Нехай на початок перекопіювання в стеку *R* міститься *n* елементів, тоді в стеку *L* знаходиться *n* + 1 елементів. Можна коректно обробити будь-яку кількість операцій push, а також *n* операцій pop. Операція empty під час перекопіювання завжди повертає false, оскільки не можна витягувати елементи з стека *L*, який не порожній. Таким чином, разом з операцією, що активує перекопіювання, гарантовано можна коректно обробити *n* + 1 операцію.

Отже, необхідні такі додаткові дії:

- перемістити вміст *R* в *S*, *n* дій;
- перемістити вміст *L* у стеки *R*, *Rc'*, *n* + 1 дій;
- перемістити перші *toCopy* елементів з *S* до *R*, *Rc'*, інші викинути, *n* дій;
- поміняти ролями стеки *Rc*, *Rc'*, *L*, *L'*, 2 дії.

Таким чином, отримали $3 \cdot n + 3$ додаткові дії за *n* + 1 операцій, або 3 = O(1) додаткові дії на операцію в режимі перекопіювання, що і вимагалося.

Тепер розглянемо, як змінилися стеки за період перекопіювання. Нехай операція empty не змінює чергу, тобто ніякі додаткові дії не здійснюються. Нехай за *n* наступних за активацією операцій, що змінюють (push, pop), надійшло *x* операцій pop, $n - x$ операцій push. Очевидно, що після перекопіювання в нових стеках виявиться: $n - x$ елементів *L*,

$$2 \cdot n + 1 - x = (n - x) + (n + 1)$$

елементів *R*, тобто до наступного перекопіювання ще *n* + 2 операції . З іншого боку, стек *Rc* містив всього n елементів, тому можна очистити його, просто видаляючи по одному елементу при кожній операції у звичайному режимі.

Отже, черга *Q* складатиметься із шести стеків *L*, *L'*, *R*, *Rc*, *Rc'*, *S*, а також двох внутрішніх змінних *recopy, toCopy*, які потрібні для коректності перекопіювання і додаткова змінна *copied*, що показує, чи переміщувалися елементи зі стеку *L* в стек *R*, щоб ці елементи не почали переміщуватися в стек *S*.

Інваріант черги (звичайний режим):

- Стек L містить ліву половину черги, порядок при виведенні зворотний.
- Стек R містить праву половину черги, порядок при вилученні прямої.
- $L.size \leqslant R.size$
- $R.size = 0 \equiv Q.size = 0$
- Rc − копія R
- $Rc'.size < R.size - L.size$
- $L'.size = 0$, $S.size = 0$

Тоді до наступного перекопіювання ($L.size = R.size + 1$) стеки L', S, Rc' гарантовано будуть порожніми.

Інваріант черги (режим перекопіювання):

- $Rc.size = toCopy$
- Якщо $L.size = 0$, то:
  - При $toCopy > 0$ перші $toCopy$ елементів S коректні, тобто дійсно містяться в черзі.
  - При $toCopy \leqslant 0$ стек R містить весь правий шматок черги у правильному порядку.

Черга працюватиме у двох режимах:

- Звичайний режим, новий елемент кладеться в L, витягається з R і з Rc, Rc' для підтримки порядку, операція empty = ($R.size = 0$).
- Режим перекопіювання, новий елемент кладеться в L', витягається з Rc, можливо з R, empty = false, потім здійснюються додаткові дії.

Також після операції у звичайному режимі слідує перевірка на активацію перекопіювання ($recopy = (L.size > R.size)$), якщо це так, то $toCopy = R.size$, $recopy$ = true, $copied$ = false, відбувається перший набір додаткових дій.

Після операції в режимі перекопіювання слідує перевірка на завершення перекопіювання ($recopy = (S.size == 0)$), а при завершенні змінюються ролями стеки Rc, Rc', L, L'.

Після того, як ми отримали чергу в реальному часі з O(1) = 6 звичайними стеками, її можна легко перетворити на персистентну, зробивши всі стеки

персистентними, але насправді персистентність дозволяє не створювати явної копії стека *R*, так що достатньо п'яти стеків.

Замість стеків *Rc, Rc′* персистентна черга зберігає один стек *R′*, в який при активації перекопіювання записується остання версія стека *R*, надалі всі операції pop звертаються саме до неї. Всі зауваження щодо *toCopy* залишаються чинними.

Також немає потреби спустошувати стек R' до моменту перекопіювання, тому що скопіювати туди нову версію *R* можна за O(1), а звільнення ділянок пам'яті безглуздо, оскільки вони використовуються в інших версіях персистентної черги.

В якості версії черги ми використовуватимемо запис

$$Q = \langle L, L', R, R', S, recopy, toCopy, copied \rangle$$

, що містить п'ять версій персистентних стеків і три змінних.

Нехай персистентний стек повертає разом із звичайним результатом роботи стека нову версію, тобто операція *S.pop* повертає ⟨*Sn*, *x*⟩, а операція *S.push(x)* повертає *Sn*.

Аналогічно свою нову версію разом із результатом операції повертає і персистентна черга, тобто *Q.pop* повертає ⟨*Qn*, *x*⟩, а *Q.push(x)* повертає *Qn*.

3.3.4 Розв’язок на чотирьох стеках

Алгоритм побудови персистентної черги на п’ятьох стеках вважається класичним, але має недолік у виглядіскладної реалізації. Тому пропонується варіант реалізації на чотирьох стеках.

Перший стек (*first*) зберігає всі елементи, коли-небудь додані до черги. При виконанні операції push елемент спочатку додається в *first*. Після перерахування лічильників заповнюється стек *fourth*.

Другий стек (*second*) підтримується таким чином, що з нього завжди можна вилучити елемент для операції pop. При виконанні операції pop елемент вилучається з *second*. Після перерахування лічильників заповнюється стек *fourth*.

Третій стек (*third*) має бути персистентною копією стека *first*. З нього беруться елементи і додаються в *fourth*.

Четвертий стек (*forth*) використовується для поповнення другого стека, забезпечуючи наявність елементів у ньому. У нього додаються деякі елементи черги, а якщо *second* стає пустим, то з нього беруться елементи для операції pop.

На рисунку 3.7 представлена схема реалізації алгоритму.

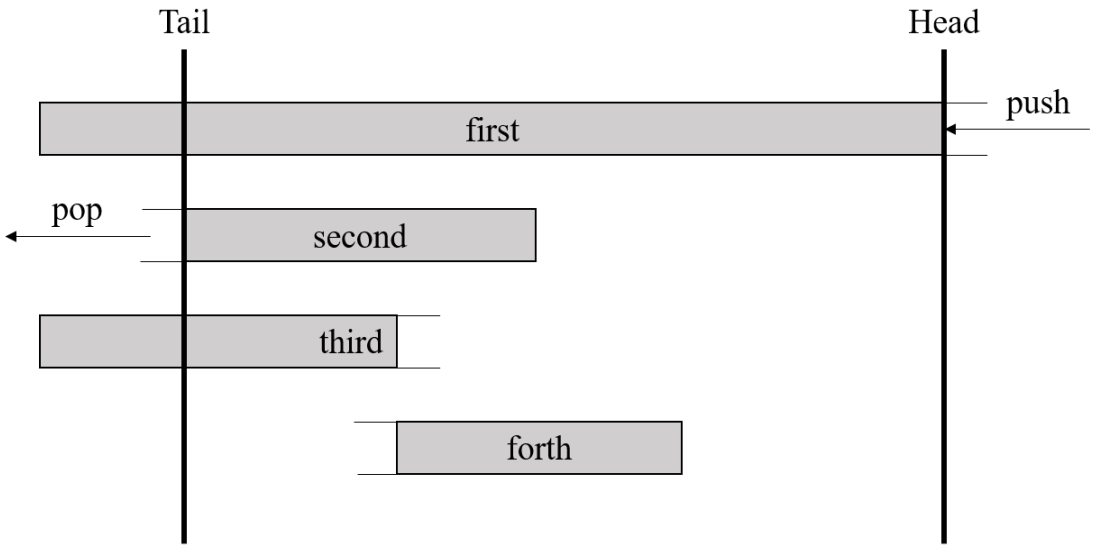

Рисунок 3.7 – Персистентна черга на чотирьох стеках

Окрім чотирьох стеків знадобляться ще два лічильники, які будуть оновлятися при заповненні стеку *fourth*. Один лічильник відповідатиме за кількість елементів у черзі (*first_size*). Другий лічильник зберігатиме кількість елементів, які треба перекинути в четвертий стек (*fourth_size*). Операція push збільшує *first_size* на одиницю і не змінює другий лічильник. Після операції pop обидва лічильники зменшуються на один.

Правила заповнення стека *fourth*:

1. Якщо *fourth_size* > 0, один елемент переміщується зі стеку *third* в *fourth,* а *third_size* зменшується на одиницю.
2. Якщо *fourth_size* = 0, стек *second* замінюється на *fourth*, а *third* і *fourth* замінюються на порожні стеки.
3. Якщо стек *fourth* порожній, стек *first* копіюється в *third* (за O(1)), а лічильники перераховуємо *fourth_size = first_size*.

Оскільки жодна з операцій не робить копію додаткових елементів, то витрати пам’яті для структури буде O(n), де n – кількість операцій додавання елемента. Асимптотичний час операцій додавання та вилучення елемента складає O(1).

Описаний алгоритм має два недоліки в порівнянні з реалізацією на п’яти або шести стеках. Головна причина їх виникнення – зберігання усіх елементів, які колись були додані, у стеку *first*.

Перший недолік – використання пам’яті. У хвості стеку *first* залишаються всі старі елементи. Якщо є обмеження на чергу, яке дозволяє використовувати не всі версії, а лише починаючи з певного номера, то оптимізувати алгоритм можна завдяки збиранню сміття, що зекономить пам’ять.

Другий недолік – при реалізації додаткових функцій, наприклад, підрахунок суми елементів поточної черги, потрібно підтримувати ще декілька додаткових стеків [7].

3.4 Персистентне дерево відрізків

Дерево відрізків – це структура даних, що будується на основі даного масиву A, і дозволяє швидко виконувати такі операції:

- change(i, x) – змінити значення A[i] на x
- F(i, j) – порахувати функцію f(A[i], A[i + 1],…, A[j]) на відрізку від i до j.

Асимптотичний час виконання базових операцій – O(log(n)). Зазвичай, функцією f є сума, мінімум або максимум на відрізку.

3.4.1 Опис вимог

Нехай існує дерево відрізків під номером 0 з усіма елементами рівними нулю. Змінна n відповідатиме за кількість дерев відрізків і на даному етапі рівна 1. Потрібно реалізувати базові операції над деревом відрізків:

- get(x, i) – зчитати елемент під номером x з версії номер i. У результаті нове дерево не створюється.

- change(x, i, y) – присвоїти елементу номер x з дерева номер i нове значення y. У результаті отримується дерево під номером n + 1.
- get_sum(i, l, r) – підрахувати суму значень на відрізку з дерева номер i, початок відрізка – елемент номер l, кінець – r. У результаті нове дерево не створюється.

3.4.2 Розв'язок

Розв'язок за допомогою методу повного копіювання полягає у тому, щоб при кожній операції change модифікацію вносити у нову копію. Такий метод є затратним плані витрат часу і пам'яті. Асимптотика операції change збільшиться до O(n), де n – кількість елементів у дереві.

Метод копіювання шляху значно оптимізує рішення задачі. Побудуємо звичайне дерево відрізків з деякими модифікаціями. У вузлі дерева зберігатимемо посилання на дочірні вершини. Для підтримання усіх версій дерева потрібно зберігати масив вузлів, які є коренями дерев. При виконанні операції change до масиву коренів буде додаватися один вузол, який буде коренем нового дерева. З нього легко можна відновити усю структуру. Перша версія персистентного дерева відрізків зображена на рисунку 3.8.

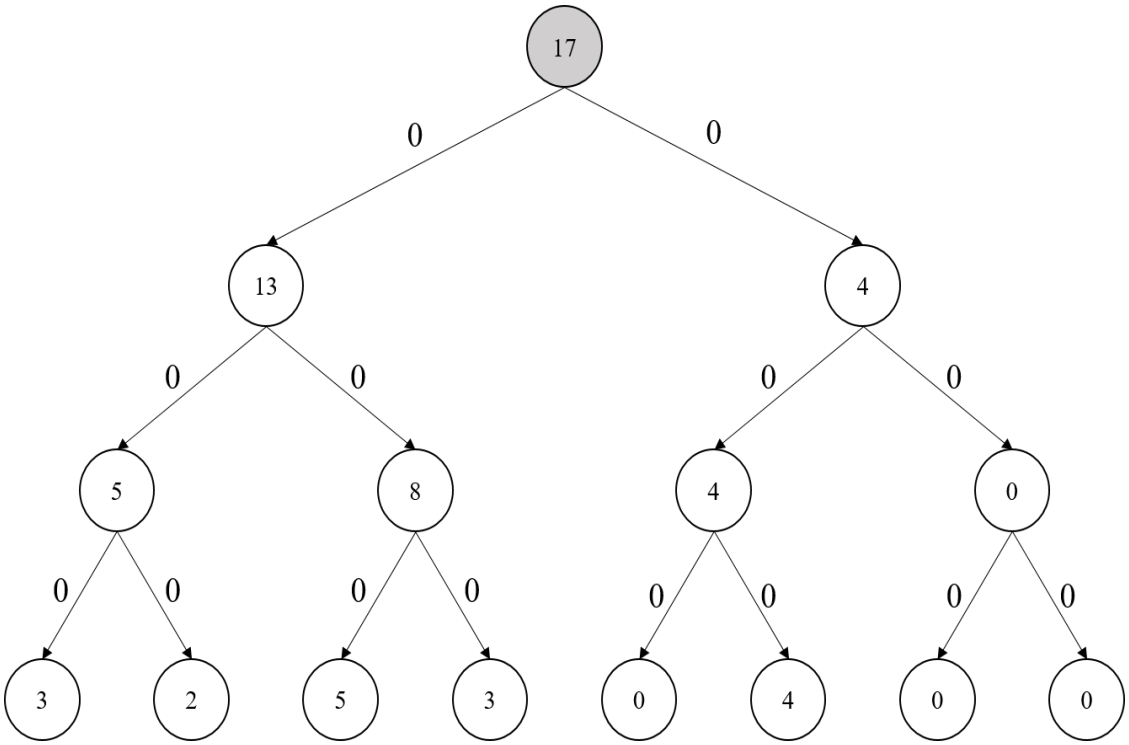

Рисунок 3.8 – Побудова персистентного дерева відрізків

Після зміни значення у деякій вершині, додаються лише ті вершини, які змінили значення або вказівники. У нових вершинах будуть зберігатися перераховані значення. Одне з посилань у нових вершинах буде вказувати на вершину зі старої версії дерева, а друге – на одну з новоутворених.

На рисунку 3.9 зображено додавання нової гілки. Алгоритм схожий з побудовою цілого дерева, де будуються дві дочірні вершини від кожної. У даному випадку потрібно будувати тільки одну у напрямі модифікованої вершини. Після цього у масив кореневих вершин додається посилання на новий корінь дерева.

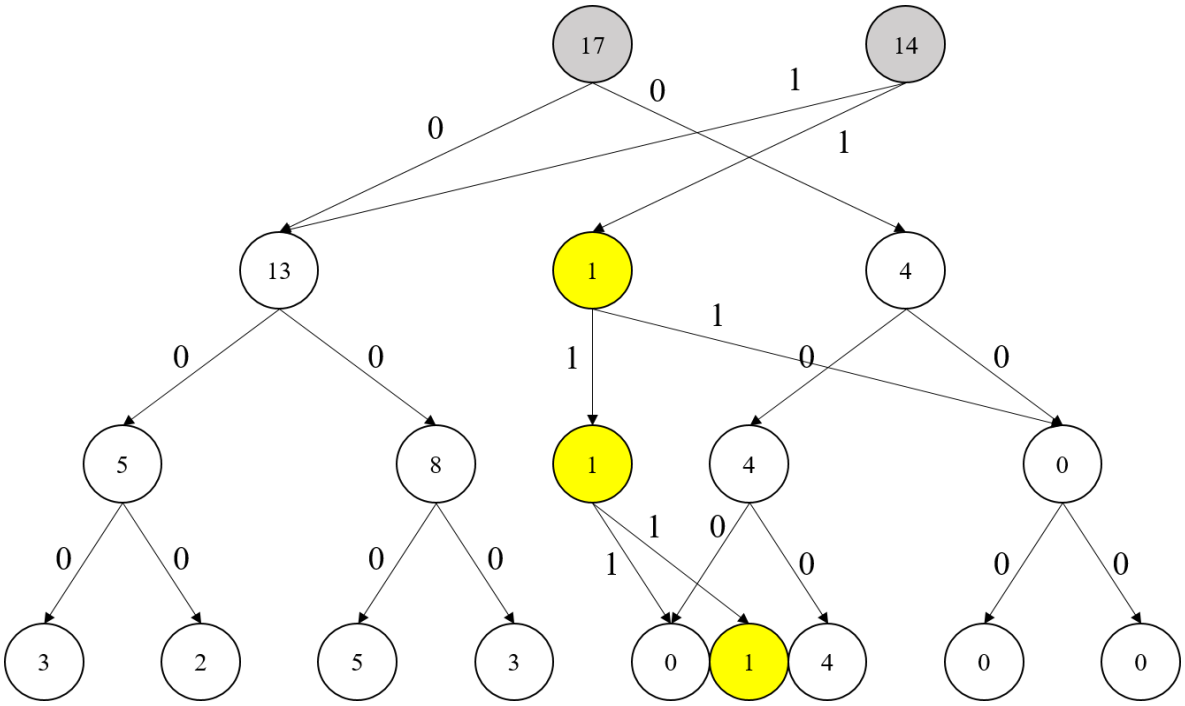

Рисунок 3.9 – Додавання нової гілки

На відміну від ефемерного дерева відрізків, при обчисленні функції F переходи відбуваються до дочірніх вершин, а запуск операції відбувається від відповідного кореня [8].

Складність побудови дерева залишається такою самою, як і в ефемерному – O(n). Єдина відмінність – додавання посилань на ліву і праву дочірні вершини, що не вплине на оцінку.

Асимптотика операції change – $O(\log(n))$, оскільки при зміні певного елементу додається $\log(n)$ вершин.

Асимптотична складність обчислення функції на відрізку - $O(\log(n))$.

Висновки до розділу 3

1. Сформовано список задач які вирішуються розробленим програмним забезпеченням.
2. Представлено алгоритм побудови повністю персистентного стеку методом копіювання шляху та наведено приклад його роботи.
3. Розглянуто варіанти побудови повністю персистентної черги на двох, чотирьох та п’яти стеках. Описано переваги та недоліки кожного з них.
4. Представлено алгоритм побудови повністю персистентного дерева відрізків з операцією суми на відрізку. Описано відмінності в порівнянні з ефемерним деревом.
5. Обчислено асимптотичну складність кожного з наведених алгоритмів, оцінено час роботи та витрати пам’яті.

4. ПРОГРАМНА РЕАЛІЗАЦІЯ ПЕРСИСТЕНТНИХ СТРУКТУР ДАНИХ

Бібліотека класів *Persistent Data Structures* розроблена на основі описаних раніше алгоритмів реалізації персистентних структур даних. Вона дозволяє працювати з повністю персистентним стеком, чергою та деревом відрізків за зручним API (Application Programing Interface).

4.1 Засоби розробки програмного продукту

У якості операційної системи для розробки програмного забезпечення було обрано Windows. Бібліотеку було реалізовано на платформі .NET 6 з використанням мови програмування C# 10.0. Коректна робота бібліотеки у проектах на нижчих версіях .NET та C# не гарантується.

Для реалізації динамічно приєднуваної бібліотеки було обрано MS Visual Studio 2022 у якості інтегрованого середовища розробки (IDE – Integrated Development Environment). Такий вибір було зроблено через те, що Visual Studio є ультимативним рішенням для розробки на C# та платформі .NET. Воно надає можливість створювати та підтримувати додатки і програмні компоненти для будь-якої операційної системи типу Windows. Створення шаблону бібліотеки реалізоване у декілька кліків миші. Так само простим є і випуск готової бібліотеки – необхідно лише зібрати проект у режимі «Release».

Програмний код було написано у парадигмі об’єктно-орієнтованого програмування.

В якості механізму для передачі та використання створеної бібліотеки було обрано NuGet, що офіційно підтримується Microsoft. NuGet – це вільна система керування пакетами, яка визначає, як створюються, розміщуються та використовуються пакети для .NET.

NuGet пакет являє собою файл з розширенням .nupkg, до складу якого входять: скомпільований код бібліотеки (DLL), описовий маніфест з характеристиками пакета та інші файли, пов’язані з кодом. Маючи код, до якого потрібно надати спільний доступ, розробники створюють пакети та публікують їх на відкритих або закритих вузлах. Споживачі за допомогою пакетного менеджера NuGet отримують їх з відповідних вузлів і додають до своїх проектів. Після цього вони мають доступ до усіх методів завантаженого пакета. При цьому NuGet сам обробляє усі проміжні дані.

Для створення власного NuGet пакету на основі скомпільованої бібліотеки було вирішено використати NuGet Package Explorer – програмне забезпечення, яке надає інтерфейс користувача для створення пакетів. Також воно дозволяє налаштувати усі можливі метадані, наприклад, власника, версію, короткий опис та підтримувані фреймворки.

4.2 Опис реалізації системи

У програмному забезпеченні, розробленому в ході виконання роботи, були реалізовані повністю персистентні стек, черга і дерево відрізків.

На рисунку 4.1 зображена діаграма класів, що представляє обрану предметну область. Структурна схема класів, які відповідають за роботу алгоритмів реалізації досліджуваних структур даних, представлена у вигляді семи класів та семи інтерфейсів:

- `INodeQueue` – інтерфейс, який описує методи та властивості звичайної черги;
- `NodeQueue` – клас, який реалізовує інтерфейс `INodeQueue`;
- `INodeStack` – інтерфейс, який описує методи та властивості звичайного стеку;
- `NodeStack` – клас, який реалізовує інтерфейс `INodeStack`;

- `IPersistentQueue` – інтерфейс, який описує методи та властивості персистентної черги;
- `PersistentQueue` – клас, який реалізовує інтерфейс `IPersistentQueue`;
- `IPersistentSegmentTree` – інтерфейс, який описує методи та властивості персистентного дерева відрізків;
- `PersistentSegmentTree` – клас, який реалізовує інтерфейс `IPersistentSegmentTree`;
- `IPersistentStack` – інтерфейс, який описує методи та властивості персистентного стеку;
- `PersistentStack` – клас, який реалізовує інтерфейс `IPersistentStack`;
- `ITreeNode` – інтерфейс, який описує властивості вузла у дереві відрізків;
- `TreeNode` – клас, що реалізовує інтерфейс `ITreeNode`;
- `IValueNode` – інтерфейс, який описує властивості вузла у стеку та черзі;
- `ValueNode` – клас, що реазізовує інтерфейс `IValueNode`.

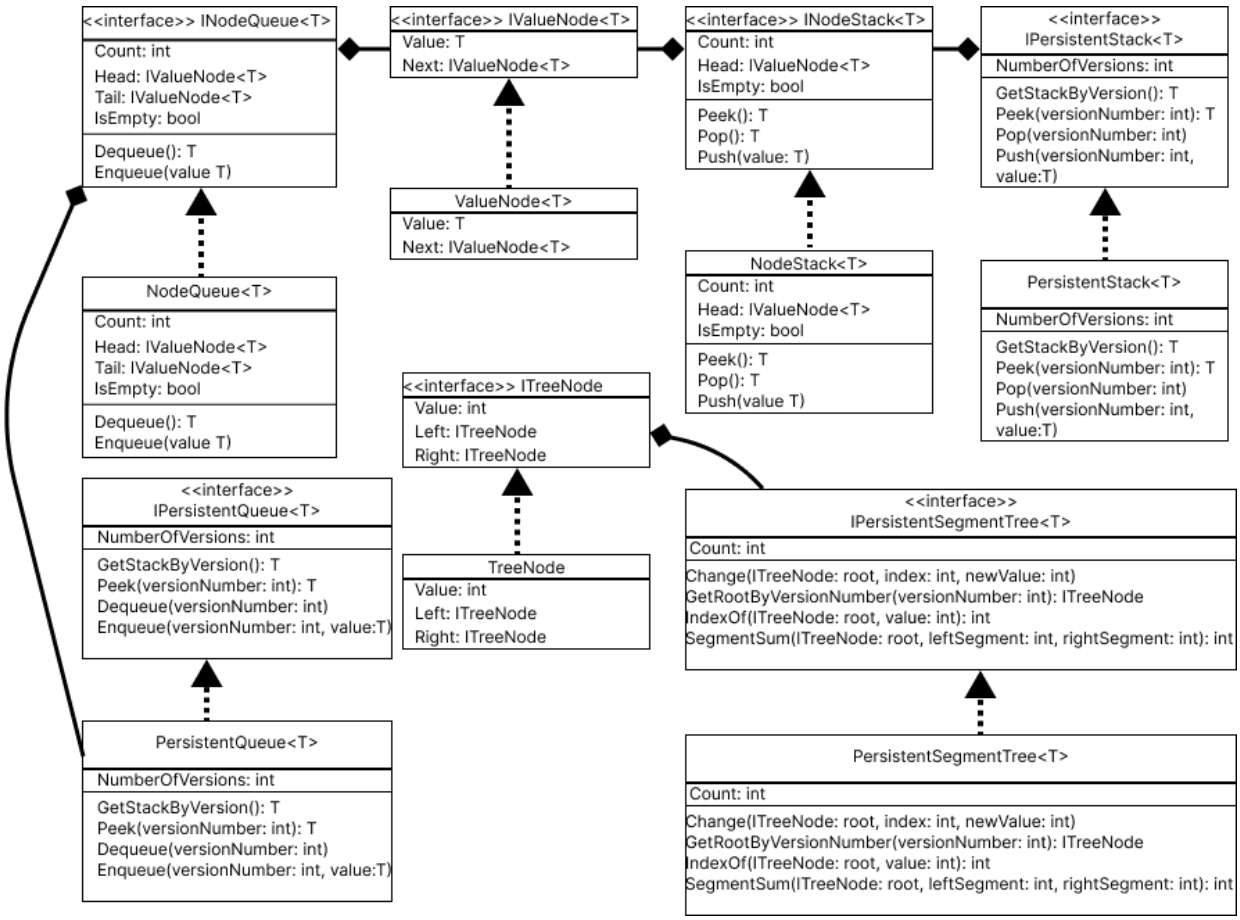

Рисунок 4.1 – Діаграма класів системи

Розглянемо кожен клас та його публічний API детальніше:

`ValueNode` має дві публічні властивості:

- `Value` – типізоване значення елементу, що зберігається у вузлі;
- `Next` – посилання на наступний вузол.

`TreeNode` має три публічні властивості:

- `Value` – цілочислове значення елементу, що зберігається у вузлі;
- `Left` – посилання на лівий дочірній вузол;
- `Right` – посилання на правий дочірній вузол.

`NodeStack` має три публічні властивості та три публічні методи

- `Head` – посилання на голову стеку;
- `Count` – цілочислове значення поточної кількості елементів у стеку;
- `IsEmpty` – логічне значення, що вказує, чи пустий стек, чи ні;
- `Peek ()` – метод, який повертає значення елементу у головному вузлі;

- `Pop ()` – метод, який вилучає головний вузол зі стеку і повертає значення елементу в ньому;
- `Push (value)` – метод, який додає новий вузол у голову стеку зі значенням `value`.

`PersistentStack` має одну публічну властивість та чотири публічні методи:

- `NumberOfVersions` – цілочислове значення максимальної кількості версій у персистентному стеку;
- `GetStackByVersion (versionNumber)` – метод, який повертає звичайний стек за номером версії;
- `Peek (versionNumber)` – метод, який повертає значення елементу у головному вузлі стеку за номером версії;
- `Pop (versionNumber)` – метод, який вилучає головний елемент з заданої номером версії стеку, формує нову версію та повертає звичайний стек, який знаходиться за номером нової версії;
- `Push (versionNumber, value)` – метод, який додає новий вузол у голову вказаного за номером версії стеку, формує нову версію та повертає звичайний стек, який знаходиться за номером нової версії.

`NodeQueue` має чотири публічних властивості та два публічні методи:

- `Head` – посилання на голову черги;
- `Tail` – посилання на хвіст черги;
- `Count` – цілочислове значення поточної кількості елементів у черзі;
- `IsEmpty` – логічне значення, що вказує, чи пуста черга, чи ні;
- `Dequeue ()` – метод, який вилучає головний вузол з черги і повертає значення у ньому;
- `Enqueue (value)` – метод, який додає новий вузол у хвіст черги зі значенням `value`.

`PersistentQueue` має одну публічну властивість та три публічні методи:

- `NumberOfVersions` – цілочислове значення максимальної кількості версій у персистентній черзі;
- `Peek (versionNumber)` – метод, який повертає значення елементу у головному вузлі черги за номером версії;
- `Dequeue (versionNumber)` – метод, який вилучає головний елемент з заданої номером версії черги, формує нову версію та повертає звичайну чергу, яка знаходиться за номером нової версії;
- `Enqueue (versionNumber, value)` – метод, який додає новий вузол у хвіст вказаної за номером версії черги, формує нову версію та повертає звичайну чергу, яка знаходиться за номером нової версії.

`PersistentSegmentTree` має одну публічну властивість та чотири публічні методи:

- `Count` – цілочислове значення максимальної кількості версій у персистентному дереві відрізків;
- `GetRootByVersionNumber (version)` – метод, який повертає кореневий вузол дерева за номером версії;
- `Change (root, index, newValue)` – метод, який змінює значення у дереві, що задане кореневим вузлом root, у вузлі, який заданий номером index, на значення `newValue`;
- `IndexOf (root, value)` – метод, який повертає індекс вузла за кореневим вузлом та номером вузла;
- `SegmentSum (root, leftSegment, rightSegment)` – метод, який обчислює суму значень на відрізку, заданому лівим і правим кінцевими індексами, у вузлах дерева, що задане кореневим вузлом.

Публікація бібліотеки до системи NuGet була здійснена за допомогою NuGet Package Explorer. Була заповнена основна інформація про збірку (автор, короткий опис, вимоги до фреймворку) та сформований пакет (рисунок 4.2).

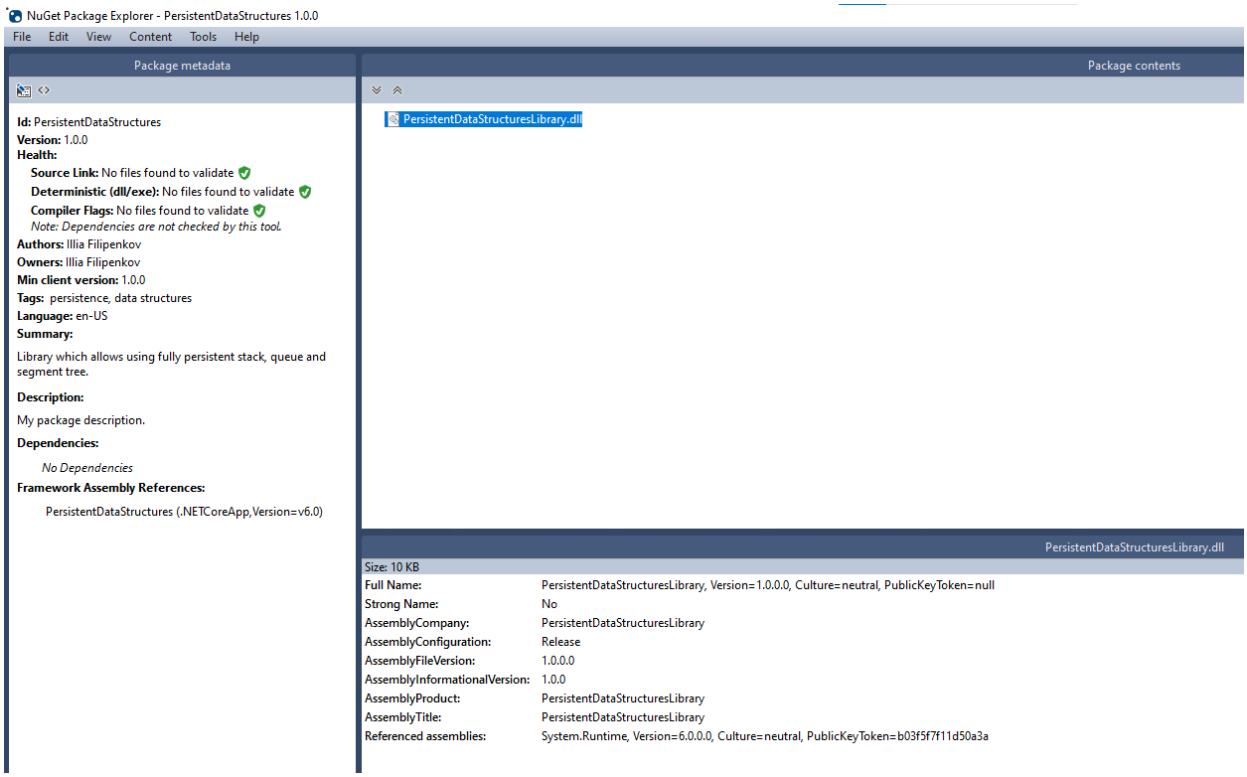

Рисунок 4.2 – Вікно NuGet Package Explorer

4.3 Оцінка використовуваних ресурсів часу та пам’яті

Для перевірки точного часу роботи персистентного стеку було створено файли з різною кількістю псевдовипадкових вхідних даних. Кожен файл з вхідними даними починається з одного числа *n* – кількості операцій. Далі слідують *n* рядків з власне операціями. Кожен рядок являє собою пару чисел, які описують операцію. Перше число – номер версії, що модифікується. Друге число – значення, яке буде додаватися у голову стеку (для операції push) або 0, якщо операція здійснює видалення голови стеку (операція pop).

Для самого генерування був розроблений алгоритм, який гарантував правильність вхідних даних. Версія для модифікації обиралася випадково, але так, щоб вона не перевищувала кількість уже згенерованих операцій. Після цього випадковим чином вибиралася операція (push або pop). Для операції push також відбувалася додаткова генерація числа від 1 до 10, яке передаватиметься до неї у якості аргумента.

Точний час виконання вираховувався за допомогою консольного додатка, який зчитував згенеровані файли та обробляв вхідні дані. Консольний додаток запускався на двох наявних конфігураціях комп'ютера:

- процесор - Intel(R) Core(TM) i7-3537U CPU 2.5 GHz, оперативна пам'ять – 8 GB (далі – Конфігурація 1)
- процесор - Intel(R) Core(TM) i3-4000M CPU 2.4 GHz, оперативна пам'ять – 8 GB (далі – Конфігурація 2)

Точний час виконання програми на різних вхідних даних на двох конфігураціях комп'ютера наведено у таблиці 4.1.

Таблиця 4.1.

Час роботи персистентного стека

Кількість операцій (n)	$2 \cdot 10^5$	$2 \cdot 10^6$	$2 \cdot 10^7$	10^8
Конфігурація 1	420 мс	4315 мс	47691 мс	247965 мс
Конфігурація 2	469 мс	5219 мс	56059 мс	301080 мс
Різниця	12%	21%	18%	21%

Для перевірки точного часу роботи персистентної черги було створено файли з різною кількістю псевдовипадкових вхідних даних. Кожен файл з вхідними даними починається з одного числа n – кількості операцій. Далі слідують n рядків з власне операціями. Кожен рядок являє собою пару чисел, які описують операцію. Перше число – номер версії, що модифікується. Друге число – значення, яке буде додаватися у кінець черги (для операції push) або 0, якщо операція здійснює видалення з початку черги (операція pop).

Для самого генерування був розроблений алгоритм, який гарантував правильність вхідних даних. Версія для модифікації обиралася випадково, але так, щоб вона не перевищувала кількість уже згенерованих операцій. Після цього випадковим чином вибиралася операція (push або pop). Для операції push також відбувалася додаткова генерація числа від 1 до 10, яке передаватиметься до неї у якості аргумента.

Точний час виконання програми на різних вхідних даних на двох конфігураціях комп’ютера наведено у таблиці 4.2.

Таблиця 4.2.

Час роботи персистентної черги

Кількість операцій (n)	10^3	10^4	10^5	10^6
Конфігурація 1	3 мс	22 мс	132 мс	2532 мс
Конфігурація 2	3 мс	26 мс	156 мс	3015 мс
Різниця	0%	18%	18%	19%

Для перевірки точного часу роботи персистентного дерева відрізків було створено файли з різною кількістю псевдовипадкових вхідних даних. Кожен файл з вхідними даними починається з двох чисел n та m – кількості елементів у масиві та кількість операцій відповідно. Далі слідують n рядків з власне операціями. Кожен рядок являє собою чотири числа, які описують операцію. Перше число – номер версії, що модифікується. Друге число – тип операції (0 – get_sum, 1 – change). Третє та четверте число – аргументи заданої операції. Для операції зміни елементу – його порядковий номер та нове значення. Операція отримання суми задається початковим та кінецевим номером відрізка, на якому вона шукається. Для оптимізації вхідних файлів, усі елементи масиву у нульовій версії рівні 0.

Для самого генерування був розроблений алгоритм, який гарантував правильність вхідних даних. Версія для модифікації обиралася випадково, але так, щоб вона не перевищувала кількість уже згенерованих операцій. Після цього випадковим чином вибиралася операція (change або sum). Для операції change відбувалася генерація номеру елементу та числа від 1 до 10, які передаватимуться до неї у якості аргумента. Для операції sum відбувалася генерація початку та кінця відрізку таких, щоб відрізок був додатнім.

Точний час виконання програми на різних вхідних даних на двох конфігураціях комп’ютера наведено у таблиці 4.3 та таблиці 4.4.

Таблиця 4.3.

Час роботи персистентного дерева відрізків на Конфігурації 1

	$m = 10^3$	$m = 10^4$	$m = 10^5$	$m = 5 \cdot 10^5$	$m = 10^6$
$n = 10^3$	4 мс	42 мс	406 мс	2500 мс	4359 мс
$n = 10^4$	6 мс	47 мс	448 мс	2377 мс	4584 мс
$n = 10^5$	5 мс	49 мс	558 мс	2431 мс	4956 мс
$n = 10^6$	4 мс	66 мс	521 мс	2605 мс	5162 мс

Таблиця 4.4.

Час роботи персистентного дерева відрізків на Конфігурації 2

	$m = 10^3$	$m = 10^4$	$m = 10^5$	$m = 5 \cdot 10^5$	$m = 10^6$
$n = 10^3$	16 мс	62 мс	440 мс	2498 мс	4796 мс
$n = 10^4$	15 мс	63 мс	484 мс	2531 мс	5172 мс
$n = 10^5$	16 мс	78 мс	579 мс	2734 мс	5515 мс
$n = 10^6$	15 мс	78 мс	609 мс	2906 мс	5874 мс

4.4 Інсталяція та системні вимоги

Інтеграція бібліотеки у інший програмний продукт відбувається за допомогою MS Visual Studio 2022. Вимоги для програми-клієнта: версія фреймворку не нижче .NET 6, версія мови – не нижче C# 10.0.

Встановлення бібліотеки можливе за двома сценаріями. Найлегший варіант – встановлення за допомогою пакетного менеджера NuGet (рисунок 4.3). У консолі пакетного менеджеру необхідно ввести команду: NuGet\Install-Package PersistentDataStructures –Version 1.0.0.

Рисунок 4.3 – Встановлення бібліотеки за допомогою консолі пакетного менеджера NuGet

Також можна встановити бібліотеку за допомогою інтерфейсу користувача для пакетів NuGet, що доступний у Visual Studio. Для цього необхідно зайти до налаштувань пакетів проекту, задати у рядок пошуку «PersistentDataStructures» та натиснути кнопку «Встановити» (рисунок 4.4).

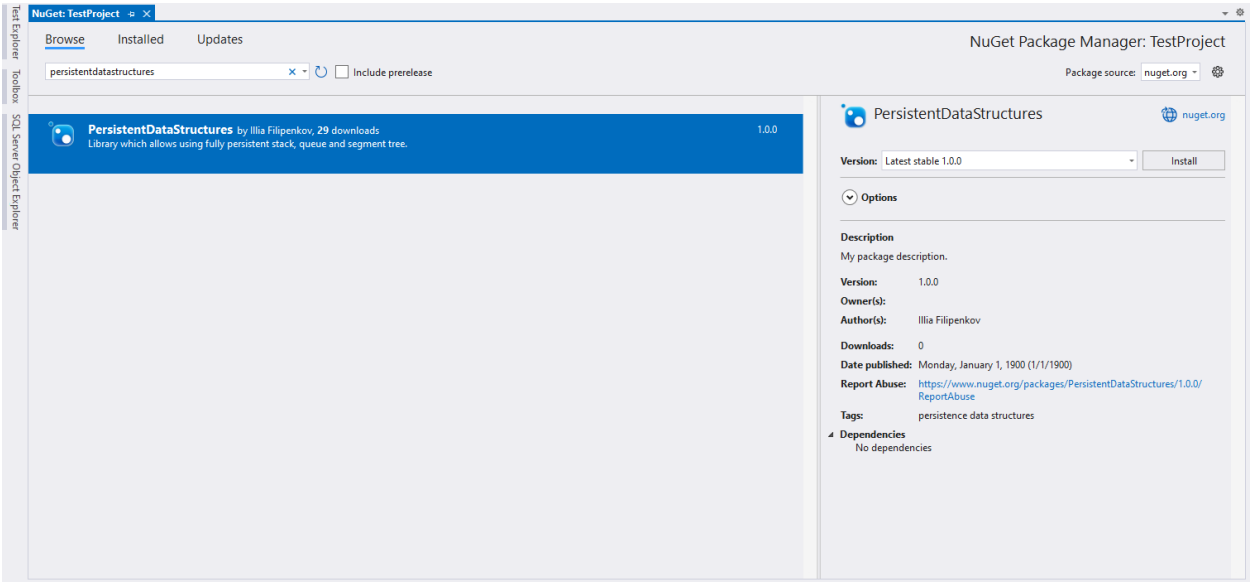

Рисунок 4.4 – Встановлення бібліотеки за допомогою інтерфейсу користувача для пакетів NuGet

Другий варіант встановлення – додавання посилання на збірку. Для цього необхідно завантажити вихідний код бібліотеки. Потім за допомогою Visual Studio

зробити збірку проекту у режимі «Release». Після цього у налаштуваннях проекту-клієнта треба встановити посилання на зроблену збірку (рисунок 4.5).

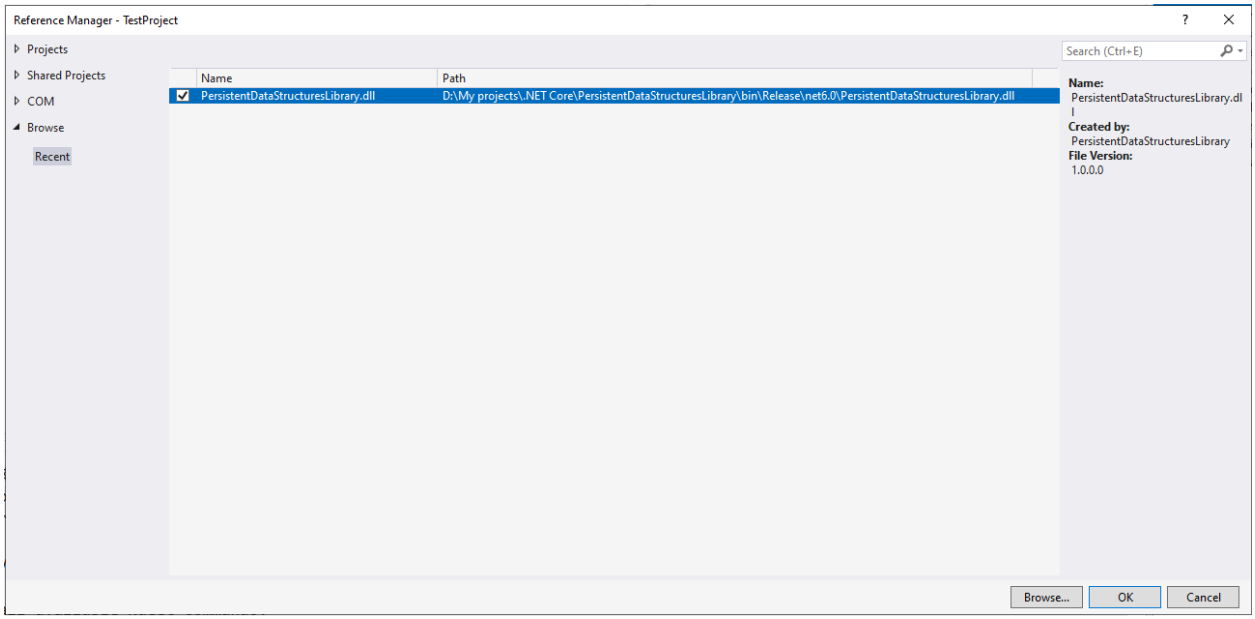

Рисунок 4.5 – Встановлення бібліотеки через додавання посилання на збірку

4.5 Деінсталяція програмної системи

Якщо бібліотеку було встановлено через систему керування пакетами NuGet, то найлегшим способом видалити її з проекту буде видалення через інтерфейс користувача у Visual Studio (рисунок 4.6).

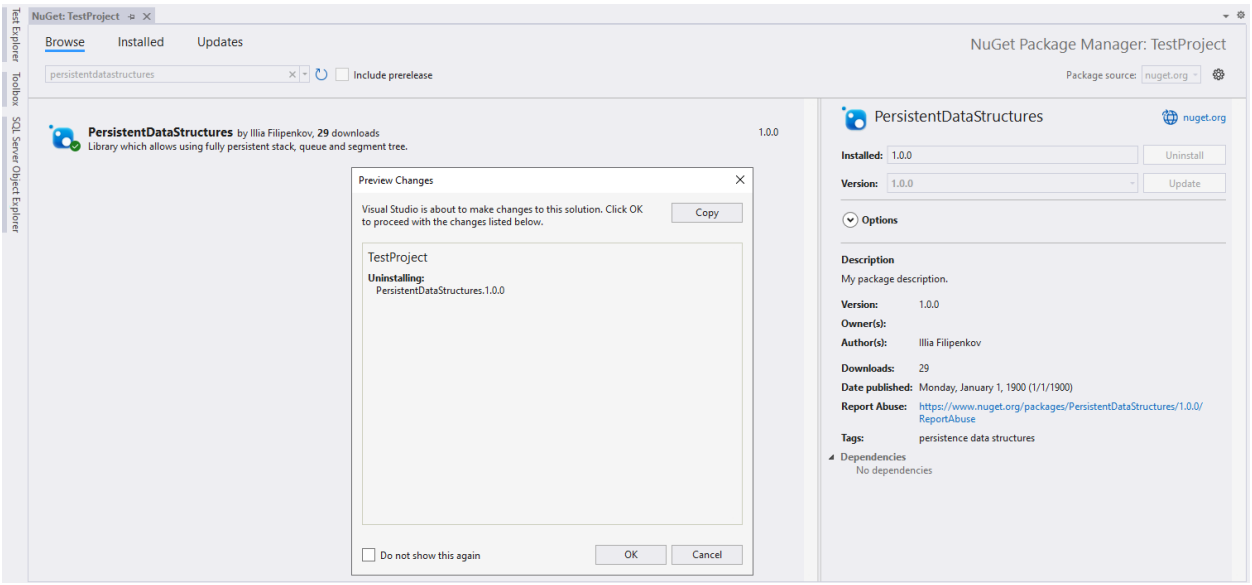

Рисунок 4.6 – Видалення бібліотеки через інтерфейс користувача NuGet

Якщо бібліотека була встановлена через посилання на збірку, необхідно у налаштуваннях проекту-клієнта зняти відмітку з цієї бібліотеки (рисунок 4.7).

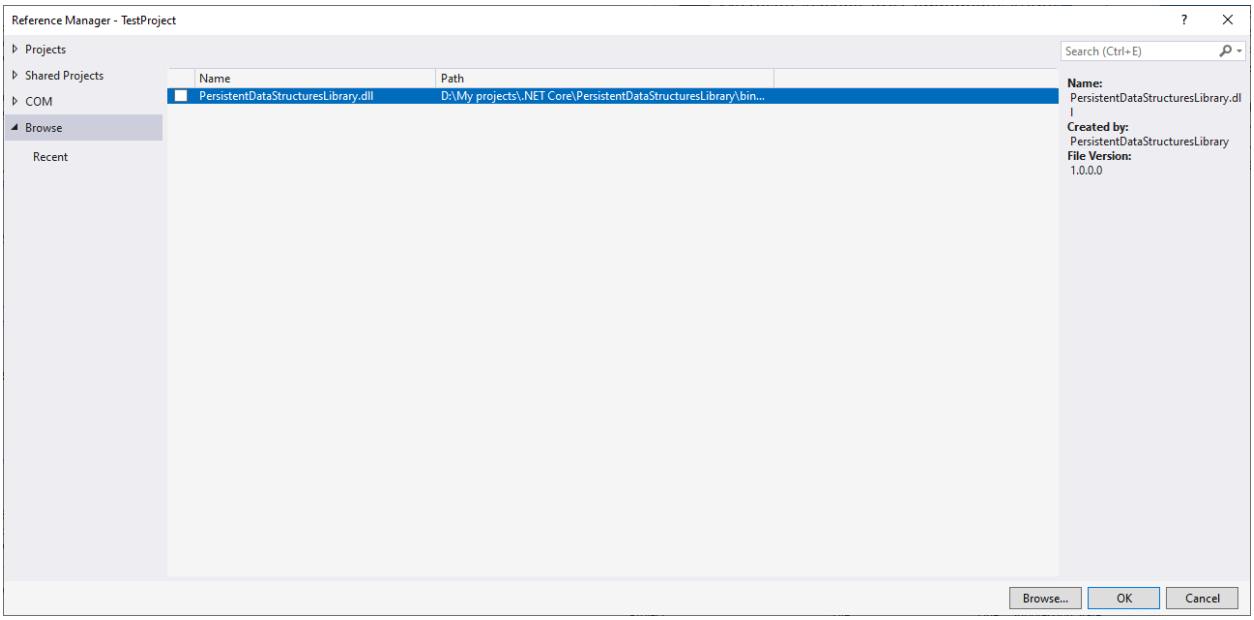

Рисунок 4.7 – Видалення посилання на збірку

Висновки до розділу 4

1. Розроблено програмне забезпечення для обробки персистентних структур даних, яке перевершує існуючі аналоги за такими характеристиками:

- реалізація персистентної черги

- можливість використання повністю персистентних структур даних
- система була розроблена мовою програмування C# з використанням принципів об'єктно-орієнтованого програмування.

2. Розроблений програмний продукт може бути впроваджений у системах контролю версіями або у технологічний комплекс, який розв'язує задачу локалізації точки з онлайн запитами.

5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ

Ідея проекту полягає у створенні бібліотеки динамічного компонування, яка надає змогу використовувати повністю персистентні структури даних у програмних системах. Такою бібліотекою зможуть користуватися інженери та розробники програмного забезпечення, які працюють над системами контролю версіями.

5.1 Опис ідеї стартап-проекту

Проаналізуємо зміст ідеї, у яких напрямках вона може бути застосована, чим вона відрізняється від вже існуючих аналогів, а також головні переваги, на які звернуть увагу потенційні користувачі. Результати аналізу представлені у таблиці 5.1.

Таблиця 5.1.

Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Створення бібліотеки динамічного компонування для обробки повністю персистентних структур даних	Розробка систем контролю версій	Використання кожної з версій структури даних як повноцінного стану системи
	Програмне забезпечення, яке розв’язує геометричні задачі локалізації точки	Зручний API, який надає доступ до обробки структур даних за оптимальними алгоритмами

На ринку існують аналоги подібних бібліотек, але саме на платформі .NET їх дуже мало. До того ж, всі вони мають обмежену функціональність, а саме, не реалізовують деякі часто вживані структури даних або реалізовують лише часткову персистентність, а не повну.

Тому необхідно провести аналіз переваг та недоліків ідеї порівняно з пропозиціями конкурентів з точки зору техніко-економічних характеристик. Порівняння наведене у таблиці 5.2.

Таблиця 5.2.

Визначення характеристик ідеї проекту

Техніко-економічні характеристики ідеї	Продукція конкурентів		Слабкі (W), нейтральні (N) та сильні (S) сторони		
Назва продукту	Immutable Collections	C5		✓	
Фреймворк	.NET Framework 1.1	.NET 5.0			✓
Розміри	25.6 Кб	974.51 Кб			✓
Необхідність підключення додаткових бібліотек	Відсутня	NUnit		✓	
Встановлення за допомогою NuGet	Ні	Так			✓

Програмне забезпечення вже розроблене та представлене у вигляді бібліотеки динамічного компонування, яка розміщена на NuGet. Розроблена

бібліотека не великого розміру (20 Кб), потребує версії фреймворку не нижче .NET 6 та версії мови C# 10.0.

Перевагами даної розробки є те, що подібні бібліотеки не реалізовують деякі структури даних, а також часто мають лише частково персистентні структури, а не повністю.

5.2 Технологічний аудит проекту

Проведення технічного аудиту ідеї проекту вимагає проведення аудиту технологій, за допомогою яких цей проект розроблявся. Результат оцінки можливості технологічної здійсненності проекту представлений у таблиці 5.3.

Таблиця 5.3.

Технологічна здійсненність ідеї проекту

Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
Створення бібліотеки динамічного компонування для використання повністю персистентних структур даних у програмних системах	Середовище розробки бібліотек динамічного компонування Visual Studio	✓	Доступно (платне)
	Пакетний менеджер NuGet Package Explorer	✓	Доступно (безкоштовне)

Обрані технології для реалізації ідеї проекту: Microsoft Visual Studio 2022 – середовище розробки бібліотек динамічного компонування та NuGet Package Explorer для налаштування та публікації пакету з бібліотекою.

Обрані технології знаходяться у вільному доступі, офіційно підтримуються Microsoft, а функціональності їхніх безкоштовних версій повністю вистачає для реалізації поставленої задачі.

5.3 Аналіз ринкових можливостей запуску стартап-проекту

Для планування напрямів розвитку проекту необхідно визначити ринкові можливості, що можуть бути використані під час ринкового впровадження проекту, та ринкових загроз, що можуть перешкодити реалізації проекту. Також потрібно врахувати стан ринкового середовища, пропозиції конкурентних проектів та потреб потенційних користувачів. У таблиці 5.4 наведено результати проведення аналізу попиту.

Таблиця 5.4.

Попередня характеристика ринку стартап-проекту

Показники стану ринку	Характеристика
Загальна потреба в продукції	Необхідна, але не є поширеною (через фінансові вигоди)
Можливі річні обсяги випуску в натуральних показниках	До 1000 копій
Ціна одиниці продукції	36$
Річні обсяги випуску в вартісних показниках	3600 – 13600$
Динаміка ринку (якісна оцінка)	Повільно зростає

Продовження таблиці 5.4.

Наявність обмежень для входу	Бажання розробників працювати лише над власним ПЗ, задля підтримки монополії у сфері
Специфічні вимоги до стандартизації та сертифікації	Для ПЗ відсутні. Для коректної роботи - використання стандартів ISO 9126 та ISO 25010
Середня норма рентабельності в галузі (або по ринку)	76%

Незважаючи на те, що ринок не є достатньо привабливим, на перший погляд, при проведенні збору статистичних даних, було визначено, що інтерес до використання нових перспективних методів зберігання і керування даними, зростає.

Отже, оскільки є потреба у розробці такого програмного забезпечення, задля полегшення роботи розробників над програмними продуктами, доцільно, необхідно, та має сенс розробка даної бібліотеки. Ця розробка цілком може зайняти не малу долю ринку.

Для планування напрямів розвитку проекту необхідно визначити ринкові можливості, що можуть бути використані під час ринкового впровадження проекту, та ринкових загроз, що можуть перешкодити реалізації проекту. Також потрібно врахувати стан ринкового середовища, пропозиції конкурентних проектів та потреб потенційних користувачів. У таблиці 5.4 наведено результати проведення аналізу попиту.

У таблиці 5.5 наведені потенційні групи користувачів, їхні особливості та виділяється перелік вимог до товару.

Після цього необхідно провести аналіз нинішньої ситуації на ринку: скласти таблиці факторів, що сприяють та перешкоджають впровадженню проекту. Результати цього аналізу наведено у таблицях 5.6 та 5.7.

Таблиця 5.5.

Характеристика потенційних клієнтів стартап-проекту

Потреба, що формує ринок	Цільова аудиторія	Особливості поведінки споживачів	Вимоги користувачів до товару
Оцінка якості розрахунку показників	Розробники програмного забезпечення для систем контролю версіями	Розробники займаються написанням програм, які не завжди: відповідають стандартам, не завжди достатньо оптимізовані, що впливає на подальше життя створених програмних продуктів – виникають проблеми, недоліки та конфлікти. Тривале вирішення проблем несумісності або критичних помилок ПЗ.	– доступна ціна; – зручність і простота впровадження; – мобільність
Оцінка якості розрахунку показників	Розробники програмного забезпечення з розв’язанням геометричних задач з локалізації точки	Ведені основною метою створити програмний продукт та випустити його на ринок, власники програмних розробок націлені на основні завдання такі, як: швидше створити ПП і якомога вигідніше продати (більше копій, вища ціна).	– зручність і простота впровадження

Таблиця 5.6.

Фактори загроз

Фактор	Зміст загрози	Можлива реакція компанії
Поява конкурентів	Можлива поява конкурентів, які спроможуться створити більш якісний продукт. Можлива поява більш дешевих продуктів	Зменшення ціни з підвищенням якості при цьому, розробка удосконалень, розширення асортименту (додавання нових можливостей, нового функціоналу та/або додання можливостей розрахунку нових параметрів)
Зміни тенденцій ринку	Можлива ситуація, в якій з'явиться більш досконала програмна система від конкурентів, які значно довше на ринку.	Майже неможлива ситуація на найближчі багато років. Але можливості вирішення найпростіші - розробка нових сучасних необхідних удосконалень, тобто додання або заміна старого функціоналу на можливості розрахунку нових параметрів
Зниження репутації компанії	Можлива ситуація, коли конкуренти спроможуться на більший попит	Зміна партнерів, заключення нових контрактів, проведення рекламних та промо-акцій
Економічний спад	Відсутність попиту на товар компанії через економічну складову	Збільшення обсягів продажів, зменшення ціни; зміна цільової аудиторії

Таблиця 5.7.

Фактори можливостей

Фактор	Зміст загрози	Можлива реакція компанії
Невелика кількість конкурентів	На ринку на сьогоднішній день дуже не значна кількість конкурентів, їхні програмні продукти в переважній більшості гірші або вузько спеціалізовані	Розповсюджувати створений продукт, розвивати його можливості
Відповідні тенденцій ринку	ІТ-ринок на сьогоднішній день потребує, а відповідно і надає всі можливості для впровадження систем, які надаватимуть користувачам можливість користуватися повністю персистентними структурами даних	Розповсюджувати створений продукт, розвивати його можливості
Можливість побудови власної репутації	Новий «гравець» на ринку має всі можливості для побудови власної репутації з «чистого листка»	Пошук замовників, можливих покупців створеного продукту, розширення бази замовників. Зарекомендувати себе, як надійну компанію. Можливо на вигідних умовах співпраці

Після цього потрібно провести ступеневий аналіз пропозиції конкурентів на ринку, визначити їхні загальні риси. У таблиці 5.8 описано тип та інтенсивність можливої конкуренції, конкурентоспроможність за різними ознаками: рівнем конкурентної боротьби, галузевою ознакою та видами товарів.

Таблиця 5.8.

Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	У чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії)
Тип конкуренції	Чиста Залежить від кількості конкурентів та якості надання ними послуг у порівнянні з послугами компанії	Покращення власного продукту через зниження ціни та підвищення якості
За рівнем конкурентної боротьби	Локальна Конкуренція на вітчизняному ринку	На вітчизняному ринку конкурентів не виявлено, а тому компанія має можливість встановлення власної бажаної ціни, та наробляти клієнтську базу. Перспектива – вихід на міжнародний рівень
За галузевою ознакою	Внутрішньогалузева Продукт націлений лише на конкретну сферу діяльності	Немає можливостей та сенсу розширювати функціонал за межі IT-сфери, але існує багато варіантів розвиватись всередині неї
Конкуренція за видами товарів	Марки-конкуренти Створений товар може мати конкурентів, які пропонують аналогічний товар	Розширення функціональних, безплатне встановлення в інститутах (задля популяризації підходу)

Продовження таблиці 5.8.

За характером конкурентних переваг	Цінова Важливо за скільки продається товар, та скільки з нього прибутку	Можливе підвищення ціни на нові розробки, зниження на старі версії для заохочення покупців у порівнянні з цінами конкурентів
За інтенсивністю	Марочна Можуть з'являтись конкуренти	На ринку цільової аудиторії поки що конкурентів не виявлено. Але при виході на міжнародний ринок потрібно рекламувати кращий функціонал створеного продукту, встановлювати конкурентоспроможні ціни, та доводити свою надійність

Модель п'яти сил М. Портера є основою для детального аналізу умов конкуренції в галузі, який слідує за аналізом конкуренції. Ця модель виділяє п'ять факторів, які впливають на привабливість ринку з точки зору характеру конкуренції:

1. Конкурент, що вже перебуває у галузі
2. Конкуренти, які потенційно можуть прийти у галузь
3. Наявність товарів-аналогів
4. Постачальники, які знаходяться в конкуренції за ринкову владу
5. Споживачі, які знаходяться в конкуренції за ринкову владу

Перелік факторів конкурентоспроможності формується на основі характеристик ідеї проекту (таблиця 5.2), вимог користувачів до товару (таблиця 5.5), факторів загроз та можливостей. Результати даного аналізу та обґрунтування факторів можна представити у вигляді таблиць (таблиця 5.9 і таблиця 5.10 відповідно).

Таблиця 5.9.

Аналіз конкуренції в галузі за М.Портером

Складові галузі	Прямі конкуренти в галузі	Потенційні конкуренти	Клієнти	Товари-замінники
	Розробники аналогічних систем	Кращі продукти, менші ціни	Мають найбільше значення. Більш важлива їх кількість, аніж постійна співпраця	Відсутні. Є лише конкуренти аналогічних розробок
Висновки	Інтенсивність конкурентної боротьби з боку прямих конкурентів незначна	Наявні усі можливості входу на ринок. Потенційні конкуренти не виявлені. Строки виходу на ринок – один день	Необхідність клієнтської-бази, тому важливо знаходити можливості приваблення споживачів до власного продукту	Немає обмежень

Таблиця 5.10.

Обґрунтування факторів конкурентоспроможності

Фактор конкурентоспроможності	Обґрунтування
Невелика кількість конкурентів на ринку	На ринку бібліотек для платформи .NET конкурентів з такою ж функціональністю немає

Продовження таблиці 5.10.

Доступність створеного продукту (програмно)	Бібліотека займає мало пам’яті на диску, не потребує підключення додаткових бібліотек, а технології, які нею підтримуються є дуже поширеними
Легкість і простота використання	Зручний зрозумілий API, створені довідка та інструкція для користувача
Відсутня потреба у постійному супроводі	Не потребує супроводу спеціалістів і постійних доробок з боку розробника
Підключення до мережі Інтернет	Не потребує підключення до мережі Інтернет після придбання продукту
Додаткові компоненти	Немає необхідності встановлення додаткових компонентів, на відміну від деяких аналогів

У таблиці 5.11 наведені результати аналізу сильних та слабких сторін стартап-проекту на основі факторів, описаних у таблиці 5.10.

Таблиця 5.11.

Порівняльний аналіз сильних та слабких сторін проекту

Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів						
		–3	–2	–1	0	+1	+2	+3
Мала кількість / відсутність конкурентів	18			✓				
Системні вимоги	10					✓		
Простота використання	18				✓			
Не потрібен супровід	10				✓			

Складання SWOT-аналізу є фінальним етапом ринкового аналізу стартап-проекту. SWOT-аналіз передбачає створення матриці аналізу сильних (Strength) та слабких (Week) сторін, загроз (Troubles) та можливостей (Opportunities) на основі попередньо проведеного аналізу. Його результати наведені у таблиці 5.12.

Таблиця 5.12.

SWOT-аналіз проекту

Сильні сторони (S):	Слабкі сторони (W):
– невелика кількість працівників; – молодий і перспективний колектив; – гнучка політика керівництва; – інноваційні технології	– брак власного устаткування; – брак робочої сили; – недостатньо оборотних коштів; – відсутність репутації компанії;
Можливості (O):	Загрози (T):
– розширення виробничої лінії; – додаткові послуги; – вихід на нові ринки; – розширення клієнтської бази; – співробітництво з іншими компаніями	– поява нових конкурентів; – зміни тенденцій попиту; – зниження репутації компанії; – економічний спад

За результатами проведеного SWOT-аналізу створюється перелік заходів ринкової поведінки для виведення стартап-проекту на ринок. Оцінюється оптимальний час, необхідний для їхньої ринкової реалізації, з урахуванням появи нових конкурентів. У таблиці 5.13 наведені альтернативи ринкової поведінки з точки зору строків та ймовірності отримання ресурсів.

Таблиця 5.13.

Альтернативи ринкового впровадження стартап-проекту

Альтернатива ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
Вихід на нові ринки	Пошук інвесторів	1-6 місяців
Розширення виробничої лінії	Пошук інвесторів	Після виходу на ринок основного продукту, до 6 місяців

Отже, для початку, варто представити ринку розроблене програмне забезпечення, а після отримання відгуків від користувачів, шукати можливості розширення програмної функціональності.

5.4 Розробка ринкової стратегії

Перший крок розробки ринкової стратегії - визначення плану охоплення ринку. Для цього у таблиці 5.14 описано цільові групи потенційних користувачів.

Таблиця 5.14.

Вибір цільових груп потенційних користувачів

Опис цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в сегменті	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
Інженери	Потребують	Попит є	Незначна	Просто
Науковці	Потребують	Попит є, проте нижчий ніж у інженерів	Незначна	Помірно

Різниця між описаними цільовими групами виявилася незначною, саме тому при використанні масового маркетингу будуть враховуватися обидві і впроваджуватися стандартизована програма.

Після визначення цільових груп сформуємо базову стратегію розвитку для роботи у певних сегментах ринку. Результати наведено у таблиці 5.15.

Таблиця 5.15.

Визначення базової стратегії розвитку

Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
Вихід на нові ринки	Стратегія диференціації	Надання товару відмінних якостей, які роблять систему особливою на фоні аналогічних розробок	Стратегія диференціації
Розширення виробничої лінії	Стратегія диференціації (допускається стратегія спеціалізації)	Надання товару кращих властивостей	Стратегія диференціації

У таблиці 5.16 визначається стратегія конкурентної поведінки за характеристиками продукту.

Таблиця 5.16.

Визначення базової конкурентної поведінки

Чи є проект «першопроходцем» на ринку	Так
Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Обидва варіанти
Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Ні
Стратегія конкурентної поведінки	Стратегія виклику лідера

Оцінивши вимоги користувачів до програмного забезпечення, розробляється стратегія позиціонування. У ній враховується стратегія конкурентної поведінки та обрана базова стратегія розвитку. Після цього формується ринкова позиція, за якою можна ідентифікувати продукт. Результати розробки стратегії позиціонування наведені у таблиці 5.17.

Таблиця 5.17.

Визначення стратегії позиціонування

Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати позицію власного проекту
Доступна ціна, простота і зручність використання	Стратегія диференціації	Вирішення важливих поставлених задач швидко, легко та зрозуміло. Доступність через ціну та технічні характеристики	– стандарти якості – метрики ПЗ – ASQAS

У результаті отримуємо план ринкової поведінки з узгодженою системою рішень, за якою визначаються напрями роботи стартап-компанії на ринку.

Проаналізувавши усі попередні результати можна сформувати таку стратегію: програмне забезпечення, що буде поширюватися, буде відмінним за властивостями від своїх аналогів, випускатися буде одна версія продукту для всіх цільових груп користувачів.

5.5 Розробка маркетингової програми стартап-проекту

Для початку необхідно сформувати маркетингову концепцію товару. У таблиці 5.18 наведені результати аналізу конкурентоспроможності продукту, на основі чого буде проводитися подальший аналіз.

Таблиця 5.18.

Визначення ключових переваг концепції потенційного товару

Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами
Оцінка якості ПП	Реалізація повністю персистентних структур даних з оптимальним часом виконання	Розрахункові показники, точність та достовірність яких можна оцінювати; кількість вхідних параметрів; самостійність програмної системи.

У таблиці 5.19 наведено трирівневу маркетингову модель товару, що включає в себе ідею, фізичні складові та особливості процесу надання продукту.

Таблиця 5.19.

Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові
Товар за задумом	Програмне забезпечення для обробки персистентних структур даних
Товар у реальному виконанні	Властивості/характеристики
	Реалізовано бібліотеку динамічного компонування для використання повністю персистентних структур даних. Програмне забезпечення опубліковане у системі розповсюдження пакетів NuGet.
Товар із підкріпленням	Якість: тестування пройшло задовільно
	До продажу: стандартна розроблена система (модуль «Modeling»)
	Після продажу: додані додаткові можливості, збільшення споживчої бази

У таблиці 5.20 проаналізовано, які цінові межі можна встановити на кінцевий продукт. Цей аналіз включає в себе рівень цін на аналогічні товари та рівень доходів споживчої групи.

Таблиця 5.20.

Визначення меж встановлення ціни

Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни
50 – 52000 $	500 – 5000 $	30 – 50 $

Після цього необхідно визначити оптимальну систему збуту. Потрібно прийняти рішення, чи проводити збут власними силами, чи залучати посередників, а також визначити оптималну глибину каналу збуту.

Таблиця 5.21.

Формування системи збуту

Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
Бажання отримати більше за менші гроші	Пошук клієнтської бази та продаж	Нульовий рівень: тільки виробник	Вертикальна маркетингова система

Фінальна складова розробки маркетингової програми – визначення концепції маркетингових комунікацій. Вона базується на специфіці поведінки користувачів та попередньо обраній основі позиціонування. Результати наведено у таблиці 5.22.

Таблиця 5.22.

Концепція маркетингових комунікацій

Поведінка цільових клієнтів	Канали комунікацій цільових клієнтів	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення
Бажання отримати більше за менші гроші	Будь-які	Низька ціна Широкий вибір функціоналу Легкий і простий у використанні продукт	Донести до користувача суть продукту, його якість, та залучити якомога більше зацікавлених клієнтів

Висновки до розділу 5

1. Було сформовано ринкову стратегію стартап-проекту:
   - створення продукту;
   - пошук потенційних користувачів;
   - у якості базової стратегії розвитку обрано стратегію диференціації. Шляхом надання користувачеві бажаного товару забезпечується конкурентоспроможність;
   - у якості стратегії конкурентної поведінки обрано стратегію виклику лідера. Вона передбачає конкурування з лідером ринку за рахунок орієнтування на усіх можливих споживачів, навіть користувачів аналогічних продуктів. Наступна ціль – обігнати лідерів обраного сегменту ринку.
2. Виходячи із результатів аналізу стану та динаміки ринку, було зроблено висновок, що у найближчій перспективі ринкове середовище буде сприятливим для впровадження розробленого програмного забезпечення.
3. Сформовано перелік переваг створеного програмного забезпечення:

- існуючі аналоги не надають тої функціональності, яка реалізована у продукті;
- попит на нові перспективні методи зберігання даних набирає популярність;
- простий у використанні API дозволяє дуже швидко інтегрувати розроблену бібліотеку у інший програмний продукт.

4. Проаналізувавши сегмент цільових користувачів програмного забезпечення, їх потреб, динаміки та рентабельності ринку, було визначено, що створений продукт є доцільним до комерціалізації.

5. Беручи то уваги бар’єри входження, потенційні групи клієнтів, стан конкуренції та конкурентоспроможності продукту, зроблено висновок, що продукт варто впроваджувати на ринок, і його розробка не буде марною.

ВИСНОВКИ

У магістерській дисертації наведено узагальнення теоретичних відомостей про персистентні структури даних. Також у роботі показано нове вирішення практичного завдання, а саме реалізації повністю персистентної черги на чотирьох стеках.

Під час вирішенні поставлених задач були отримані такі результати:

1. Незважаючи на те, що часткова персистентність зазвичай досягається доволі просто для будь-яких структур даних, повна персистентність може бути складною в реалізації і досягатися зовсім іншим алгоритмом.

2. Метод копіювання шляху дозволяє зберегти значну кількість пам’яті в порівнянні з повним копіюванням, оскільки копіює лише зв’язані з модифікованим вузли. Однак, часто цей підхід не є найоптимальнішим.

3. Метод «товстих вузлів» зводить витрати пам’яті до мінімуму і є застосовним для усіх структур, які можна представити у моделі «машини вказівників». Недоліки цього методу виявляються, коли створюється багато версій і виникає необхідність обходу всієї структури.

4. Якщо задача з перетворення структури даних на персистентну не вирішується за допомогою вищезгаданих методів, але структура даних може бути представлена у вигляді «машини вказівників», то використовується комбінований метод, який є складнішим у реалізації.

5. Враховуючи особливості поставленої задачі, для визначення можливостей вирішення задачі були проаналізовані існуючі програмні продукти, а саме бібліотеки ImmutableCollections та C5.

6. Проаналізовано недоліки цих бібліотек та обґрунтована необхідність розробки нової бібліотеки.

7. Сформовано перелік задач які мають вирішуватись програмним забезпеченням.

8. Представлено алгоритм побудови повністю персистентного стеку методом копіювання шляху та наведено приклад його роботи.

9. Розглянуто варіанти побудови повністю персистентної черги на двох, чотирьох та п’яти стеках. Описано переваги та недоліки кожного з них.

10. Представлено алгоритм побудови повністю персистентного дерева відрізків з операцією суми на відрізку. Описано відмінності в порівнянні з ефемерним деревом.

11. Обчислено асимптотичну складність кожного з наведених алгоритмів, оцінено час роботи та витрати пам’яті.

12. Розроблено програмне забезпечення для обробки персистентних структур даних, яке перевершує існуючі аналоги за такими характеристиками:

- реалізація персистентної черги
- можливість використання повністю персистентних структур даних
- система реалізована сучасною мовою програмування C#, яка заснована на принципах об’єктно-орієнтованого програмування.

13. Розроблений програмний продукт може бути впроваджений у системах контролю версіями або у технологічний комплекс, який розв'язує задачу локалізації точки з онлайн запитами.

Результати дипломної роботи, а саме програмне забезпечення – бібліотека динамічного компонування для обробки персистентних структур даних, а також документація програмного супроводу були використані в розробках спеціалізованого програмного забезпечення ТОВ «Свіалком».

Подальший розвиток виконаної розробки полягає в реалізації більшої кількості структур даних та додаткових допоміжних функцій над ними.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. James R. Driscoll Making Data Structures Persistent / James R. Driscoll, Neil Sarnak, Daniel D. Sleator, Robert E. Tarjan // Journal of computer and system sciences. – 1989. - №38. – p. 86-124.
2. Маврин П. Ю. Дополнительные главы алгоритмов, часть 1: лекция №13 / Маврин П. Ю. // Computer Science Center. – 2019.
3. Соколов Б. С. Персистентные структуры данных и их эффективная реализация [Текст] / Соколов Б. С. – М.: Изд-во. Мир, - 1990 – С. 235.
4. Алгоритми і структури даних [Електронний ресурс] : опорн. консп. лекцій / уклад. В. І. Манжула. - Тернопіль, 2015. - 63 с.
5. David Karger Advanced Algorithms Lecture 2: September 9, 2005 / David Karger. – 2005.
6. Haim Kaplan Persistent Data Structures / Haim Kaplan // Handbook on Data Structures and Applications. – 2001.
7. Sylvain Conchon Semi-persistent Data Structures / Sylvain Conchon, Jean-Cristophe Filliatre // Programming Languages and Systems, Lecture Notes in Computer Science, vol. 4960, Springer Berlin Heidelberg. – 2008. – p. 322-336.
8. Bagwell Tiark RRB-Trees: Efficient Immutable Vectors / Bagwell Tiark, Philip Rompf. – 2011.
9. Gerth Stolting Brodal Purely Dunctional Worst Case Constant Time Catenable Sorted Lists / Gerth Stolting Brodal, Christos Makris, Kostas Tsichlas // Lecture Notes in Computer Science, Springer Berlin Heidelberg. – 2006. – p. 172-183.
10. Neil Sarnak Planar Point Location Using Persistent Search Trees / Neil Sarnak, Robert E. Tarjan // Communiations of the ACM. – 1986.
11. Olle Liljenzin Confluently Persistent Sets and Maps. – 2013.
12. Phil Bagwell Ideal Hash Trees / Technical Report. – October 2001.
13. Michael J. Steindorfer Optimizing Hash-Array Mapped Tries for Fast and Lean Immutable JVM Collections / ACM Sigplan Notices. – 2015. – p. 783-800.

14. Rich Hickey Are We There Yet? // JVM Language Summit. – 2009.

15. Robert T. Hood Real Time Queue Operations in Pure LISP / Robert T. Hood, Robert C. Melville. – Cornell University. – 1980.

16. Jacob E. Goodman Handbook of Discrete and Computational Geometry / Jacob E. Goodman, Joseph O'Rourke, Csaba D. Toth. – 2017.

17. Mark Berg Computational Geometry / Mark Berg, Otfried Cheong, Marc Kreveld, Mark Overmars. – 2008.

18. Brian Cloteaux Some Separation Results Between Classes of Pointer Algorithms / Brian Cloteaux, Desh Ranjan. – 2006.

19. Mi Young Lee Method of Providing Persistence to Object in C++ Object Oriented Programming System / Mi Young Lee, Ok Ja Cho, Dae Young Hur. – 1999.

20. Bernard Chazelle Filtering Search: A New Approach to Query-Answering // SIAM J. Computing. – 1986. – p. 703-724.

21. Bernard Chazelle Fractional Cascading: A Data Structuring Technique / Bernard Chazelle, Leonidas J. Guibas // Algorithmica. – 1986. – p.133-162.

22. Richard Cole Searching and storing similar lists // Journal of Algorithms. – 1986. – p. 202-220.

23. Chris Okasaki Purely Functional Data Structures. – Cambridge University Press. – 1999.

24. Thejaka Kanewala Persistent data structure library for C++ applications / Thejaka Kanewala Sanath Jayasena. – 2010.

ДОДАТОК А
АКТ ВПРОВАДЖЕННЯ

«Затверджую»
Директор ТОВ «Свіалком»
Богдан ТИТАРЧУК

АКТ ВПРОВАДЖЕННЯ

результатів дипломної роботи спеціаліста Філіпенкова І. Г. на тему «Програмне забезпечення обробки персистентних структур даних», яка виконана в Національному технічному університеті України «Київський політехнічний інститут імені Ігоря Сікорського»
20 листопада 2022 р.

Нами, представниками кафедри цифрових технологій в енергетиці НН ІАТЕ КПІ ім. Ігоря Сікорського та ТОВ «Свіалком», даний акт складено про те, що для використання в розробках спеціалізованого програмного забезпечення ТОВ «Свіалком» прийняті результати дипломної роботи магістра Філіпенкова І. Г., а саме програмне забезпечення – бібліотека динамічного компонування для обробки персистентних структур даних, а також документацію програмного супроводу.

Представник кафедри цифрових технологій в енергетиці НТУУ «КПІ ім. І. Сікорського»

Керівник дипломної роботи _____ Юлія СИДОРЕНКО

Представник ТОВ «Свіалком»

Директор _____ Богдан ТИТАРЧУК