Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет Комп'ютерні технології та системи
Кафедра Комп'ютерні інформаційні технології

Пояснювальна записка

до кваліфікаційної роботи
ОС магістр

на тему: «Дослідження часової ефективності підсистеми динамічної пам'яті в ОС Windows»

за освітньою програмою 12 Інженерія програмного забезпечення
зі спеціальності: 121 Інженерія програмного забезпечення

Виконав: студент групи ПЗ2321: _____ / Максим ЛИСЕНКО /

Керівник: _____ /Вадим АНДРЮЩЕНКО /

Нормоконтролер: _____ / Світлана ВОЛКОВА /

Засвідчую, що у цій роботі немає
запозичень з праць інших авторів
без відповідних посилань.

Студент _____

Дніпро – 2025

Ministry of Education and Science of Ukraine
Ukrainian State University of Science and Technologies

Faculty Computer technologies and systems
Department Computer information technology

Explanatory Note

to Master's Thesis

on the topic: «Study of the time efficiency of working with dynamic memory in the Windows OS»

according to educational curriculum **12 software engineering**
in the Speciality: **121 software engineering**

Done by the student of the group PZ2321: _____ / Maxim LYSENKO /

Scientific Supervisor: _____ / Vadym ANDRYUSCHENKO/

Normative controller: _____ / Svitlana VOLKOVA /

Dnipro – 2025

Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет: Комп'ютерних технологій і систем

Кафедра: Комп’ютерні інформаційні технології

Рівень вищої освіти: магістр

Освітня програма: **12** Інженерія програмного забезпечення

Спеціальність: **121** Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ____КІТ
_____ Вадим ГОРЯЧКІН
_____ _____ 202__ р.

ЗАВДАННЯ

На кваліфікаційну роботу ____Магістр_____

студенту Лисенко Максиму Олександровичу

1. Тема дипломної роботи: Дослідження часової ефективності роботи з динамічною пам'яттю в ОС Windows.
   Керівник роботи: Андрющенко Вадим Олександрович
   затверджені наказом 1186 ст від 29.12.2023 року
2. Строк подання студентом роботи 20.12.2024 року
3. Вихідні дані до дипломної роботи: програмний продукт.
4. Зміст пояснювальної записки (перелік питань до розробки):
   4.1. Аналітична частина: огляд предметної галузі;
   4.2. Основна частина: опис моделі, опис прототипів, опис методів дослідження;
   4.3. Експеримент та висновки.
5. Перелік демонстраційного матеріалу:
   5.1. презентація;
   5.2. демонстраційне відео.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів	Примітка
1	Вступ	15.05.2024	
2	Аналіз сучасного стану дослідження проблеми за науковими літературними джерелами	01.06.2024	
3	Аналіз сучасного стану програмно-апаратного забезпечення, яке потребує вдосконалення для вирішення проблем дослідження	01.07.2024	
4	Постановка задачі, технічне завдання	01.08.2024	30%
5	Техніко-економічні показники	01.09.2024	
6	Розробка інструментальних засобів дослідження	12.09.2024	
7	Виконання досліджень	01.10.2024	60%
8	Оформлення тез доповідей	15.10.2024	
9	Оформлення статті у фаховий журнал	01.11.2024	
10	Оформлення пояснювальної записки	15.11.2024	
11	Розробка демонстраційних матеріалів	18.12.2024	100%
12	Подання кваліфікаційної роботи до кафедри	20.12.2024	
13	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії		

Студент _____ /Максим ЛИСЕНКО/

Керівник роботи _____ /Вадим АНДРЮЩЕНКО/

РЕФЕРАТ

Ліве поле- 3.0 по всі роботі!

Магістерська кваліфікаційна робота «Дослідження часової ефективності роботи з динамічною пам'яттю в ОС Windows»: 133 с., 54 рис., 12 табл., 30 літературних джерел.

Метою роботи є дослідження часової ефективності роботи з динамічною пам'яттю в операційній системі Windows шляхом аналізу основних операцій виділення та звільнення пам'яті, а також визначення впливу різних факторів на продуктивність системи.

У роботі вирішено задачу побудови системи для дослідження часової ефективності операцій з динамічною пам'яттю в ОС Windows, яка дозволяє проводити детальний аналіз операцій з пам'яттю, включаючи динамічні структури, роботу зі словниками, маніпуляції зі стрічками, небезпечний код і покажчики, а також фрагментацію пам'яті. Система забезпечує можливість вимірювання таких параметрів, як час виконання, використання пам'яті та навантаження на процесор під час роботи з великими обсягами даних.

Розроблено алгоритми та програмне забезпечення для проведення тестів з динамічною пам'яттю, що дозволяє досліджувати часову ефективність операцій у різних умовах. Програма автоматизує процес збору даних щодо ефективності роботи з динамічними структурами і надає можливість детально аналізувати отримані результати з метою оптимізації управління пам'яттю в ОС Windows.

Розроблена методика та програмне забезпечення відносяться до галузі системного програмування та можуть бути використані для підвищення продуктивності програм, що працюють з великими обсягами динамічних даних у середовищі Windows.

ЧАСОВА ЕФЕКТИВНІСТЬ, УПРАВЛІННЯ ПАМ'ЯТТЮ, ДИНАМІЧНІ СТРУКТУРИ, ФРАГМЕНТАЦІЯ ПАМ'ЯТІ, ОПЕРАЦІЙНА СИСТЕМА WINDOWS, НЕБЕЗПЕЧНИЙ КОД, ПОКАЖЧИКИ, СЛОВНИКИ, СТРИЧКИ, ВИДІЛЕННЯ ПАМ'ЯТІ, СИСТЕМНЕ ПРОГРАМУВАННЯ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАК, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

Алокація пам'яті – процес виділення певного обсягу пам'яті для збереження даних або виконання операцій

Динамічна пам'ять – пам'ять, яка виділяється або звільняється під час виконання програми на вимогу

Небезпечний код – код, що здійснює прямий доступ до пам'яті або ресурсів без використання безпечних абстракцій, як-от покажчики в C++

Покажчики – змінні, що зберігають адреси інших змінних або об'єктів у пам'яті

Словник – структура даних, що зберігає пари ключ-значення для швидкого доступу до даних

Сторінковий файл – файл на жорсткому диску, що використовується для розширення оперативної пам'яті через механізм віртуальної пам'яті

Стрічки – тип даних, що представляє собою послідовність символів, або текстові рядки

Фрагмент пам'яті – частина виділеної пам'яті, що використовується для зберігання даних або виконання операцій

Фрагментація пам'яті – процес, при якому доступна пам'ять розбивається на невеликі блоки, що ускладнює або уповільнює процес виділення пам'яті

CPU – Центральний процесор

RAM – Оперативна пам'ять

VirtualAlloc – Функція виділення віртуальної пам'яті в ОС Windows

VirtualFree – Функція звільнення віртуальної пам'яті в ОС Windows

ВСТУП

Сучасні операційні системи є ключовим компонентом будь-якої обчислювальної інфраструктури, забезпечуючи ефективне управління ресурсами комп'ютера. Одним із найважливіших аспектів їхньої роботи є керування пам'яттю, що впливає на продуктивність додатків та загальну ефективність системи. У цьому контексті динамічне керування пам'яттю відіграє центральну роль, оскільки саме воно визначає, як програми отримують доступ до необхідних ресурсів, як відбувається виділення та звільнення пам'яті під час виконання завдань. Цей процес є надзвичайно важливим для багатьох сучасних додатків, особливо тих, що працюють у реальному часі, обробляють великі обсяги даних або вимагають швидкої реакції на події [1].

Проблема оптимального керування динамічною пам'яттю в операційних системах була об'єктом досліджень упродовж багатьох десятиліть. Існує багато підходів до її розв'язання, проте виклики залишаються. Основні труднощі полягають у балансуванні між швидкістю доступу до пам'яті та мінімізацією фрагментації, що може призвести до неефективного використання ресурсів. У випадку Windows, як однієї з найпоширеніших операційних систем, ці питання є особливо актуальними, адже вона використовується як у домашніх середовищах, так і на підприємствах, які вимагають високої надійності та продуктивності [2].

Розробка ефективних алгоритмів керування динамічною пам'яттю ускладнюється тим, що різні типи додатків мають свої специфічні вимоги до продуктивності. Зокрема, для деяких програм важлива швидкість виділення та звільнення пам'яті, тоді як для інших ключовим аспектом є мінімізація затримок при фрагментації. Таким чином, дослідження ефективності роботи з динамічною пам'яттю в операційній системі повинно враховувати специфіку різних типів навантаження та аналізувати, які методи та алгоритми виявляються найкращими у різних сценаріях використання.

Сучасні наукові дослідження приділяють значну увагу аналізу різних підходів до керування динамічною пам'яттю, таких як використання хеш-таблиць, дерев

пошуку, алгоритмів на основі списків вільних блоків пам'яті тощо. Проте ці рішення мають свої обмеження та недоліки, які можуть впливати на загальну ефективність системи. У зв'язку з цим необхідним є подальший аналіз та оптимізація існуючих методів для того, щоб досягти більшої продуктивності та стійкості операційної системи під час високих навантажень [3].

Актуальність теми дослідження полягає у необхідності пошуку оптимальних рішень для покращення роботи з динамічною пам'яттю в операційній системі Windows, яка, завдяки своїй поширеності, є важливим об'єктом для вивчення. Критичний аналіз існуючих підходів та їхнє порівняння дає змогу виявити недоліки та можливості для вдосконалення. Наприклад, використання стандартних механізмів виділення пам'яті у Windows, таких як HeapAlloc, може призводити до неефективного використання ресурсів у певних ситуаціях, зокрема під час роботи з великими обсягами даних або в реальному часі.

Для України, як держави, що активно впроваджує інформаційні технології у всі сфери економіки та державного управління, питання продуктивного використання обчислювальних ресурсів має стратегічне значення. Оптимізація систем керування пам'яттю дозволяє знижувати витрати на обслуговування інфраструктури, підвищувати надійність і безперебійність роботи критично важливих систем, а також забезпечувати конкурентоспроможність українських компаній на міжнародному ринку. Окрім того, дослідження у цій сфері сприяють розвитку вітчизняної науки та технологій, зокрема у галузі комп'ютерних наук, де Україні необхідно постійно зміцнювати свої позиції.

Об'єктом дослідження є процес управління динамічною пам'яттю в операційній системі Windows, який включає виділення, звільнення та оптимізацію використання пам'яті під час виконання програм.

Предметом дослідження є часова ефективність реалізації та використання методів управління динамічною пам'яттю в ОС Windows, що включає порівняння різних підходів до виділення і звільнення пам'яті та їхній вплив на продуктивність системи.

Метою роботи є дослідження часової ефективності роботи з динамічною пам'яттю в операційній системі Windows шляхом аналізу основних операцій виділення та звільнення пам'яті, а також визначення впливу різних факторів на продуктивність системи.

Для досягнення цієї мети необхідно вирішити такі основні задачі:

– проаналізувати поняття часової ефективності та дослідити приклади її використання в контексті управління динамічною пам'яттю в ОС Windows;

– вивчити та порівняти різні методи виділення та звільнення пам'яті, які використовуються в операційній системі Windows, з урахуванням їх впливу на продуктивність;

– провести огляд існуючих програмних рішень для тестування ефективності управління пам'яттю та вибрати оптимальні інструменти для дослідження.

– розробити програмне забезпечення для тестування часової ефективності управління динамічною пам'яттю, провести його тестування та аналіз результатів з метою виявлення шляхів оптимізації.

Для досягнення поставленої мети в роботі були використані такі методи дослідження:

– аналіз літературних джерел та наукових праць – застосовувався для вивчення теоретичних аспектів управління динамічною пам'яттю в операційних системах, зокрема ОС Windows, а також для дослідження поняття часової ефективності та різних підходів до оптимізації пам'яті;

– моделювання – застосовувалося для проектування імітаційних моделей тестування ефективності управління пам'яттю, що дозволило відтворити різні сценарії використання динамічної пам'яті та оцінити їхню ефективність;

– експериментальний метод – полягав у розробці програмного забезпечення для тестування часової ефективності роботи з пам'яттю, проведенні тестувань у різних умовах навантаження та отриманні даних щодо продуктивності різних методів управління пам'яттю;

– метод статистичної обробки даних – використовувався для аналізу результатів експериментів, зокрема обчислення середнього часу виконання операцій з пам'яттю, мінімізації фрагментації та впливу на загальну продуктивність системи.

Практичне значення одержаних результатів полягає в тому, що вони можуть бути використані для оптимізації управління пам'яттю в різних програмних середовищах, особливо в операційній системі Windows. Розроблене програмне забезпечення дозволяє здійснювати детальний аналіз часової ефективності різних операцій з динамічною пам'яттю, таких як виділення, доступ та звільнення пам'яті. Це сприяє покращенню продуктивності систем за рахунок вибору оптимальних методів для управління пам'яттю у конкретних умовах експлуатації.

Отримані результати також можуть бути впроваджені в процеси розробки програмного забезпечення, що працює з великими масивами даних або виконує складні операції з динамічними структурами, пов'язаними списками та асоціативними структурами. Це дозволить розробникам краще оцінювати та контролювати використання ресурсів, зокрема пам'яті, процесорного часу та запобігати проблемам, пов'язаним із фрагментацією пам'яті.

Окрім цього, результати дослідження можуть бути корисними для оптимізації алгоритмів обробки даних, що потребують високої швидкодії, таких як системи реального часу або обчислювальні системи з великим навантаженням на пам'ять. Аналіз фрагментації та використання небезпечного коду для доступу до пам'яті відкриває нові можливості для підвищення ефективності роботи критичних додатків. Впровадження цих оптимізацій може знизити використання ресурсів та підвищити загальну продуктивність системи, що є важливим у контексті масштабних програмних рішень для бізнесу та промисловості, зокрема для українських компаній.

1 ПОНЯТТЯ ТА МЕТОДИ ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ УПРАВЛІННЯ ДИНАМІЧНОЮ ПАМ'ЯТТЮ В ОС WINDOWS

1.1 Поняття часової ефективності та приклади використання

Часова ефективність є одним із ключових параметрів, який використовується для оцінки продуктивності програмного забезпечення та систем. Вона характеризує, наскільки швидко програма або система виконує свої функції за визначений проміжок часу. Особливе значення цей параметр набуває у задачах, що потребують високої швидкодії, таких як системи реального часу, великі обчислювальні системи та ресурсоємні додатки [4].

На рис. 1.1 представлена діаграма, що ілюструє основні параметри оцінки часової ефективності.

Рисунок 1.1 – Основні параметри оцінки часової ефективності

Часова ефективність може бути оцінена за допомогою кількох показників, серед яких найважливіші:

– час виконання – тривалість, необхідна для виконання певної операції або набору операцій. Це може бути загальний час виконання програми або час на виконання конкретної операції, наприклад, виділення пам'яті, додавання елементів до масиву чи звільнення пам'яті;

– затримки при доступі до ресурсів – це проміжки часу, необхідні для доступу до пам'яті або інших системних ресурсів. Затримки можуть бути викликані фрагментацією пам'яті, нестачею ресурсів або конкурентним доступом до них;

– навантаження на процесор – вимірюється в обсязі використовуваних процесорних циклів для виконання певних операцій. Високе процесорне навантаження може вказувати на те, що обраний метод є неефективним або потребує оптимізації;

– використання пам'яті – хоча цей показник стосується більше просторової ефективності, але його вплив на часову ефективність є значним, оскільки надмірне використання пам'яті може призводити до збільшення часу доступу до неї через фрагментацію або перевантаження [5].

Управління пам'яттю – один із найбільш важливих аспектів функціонування операційної системи, особливо в таких операціях, як виділення, звільнення, та доступ до пам'яті. Часова ефективність операцій із пам'яттю безпосередньо впливає на загальну продуктивність системи та виконання програм.

Операційна система надає кілька способів для організації керування пам'яттю, серед яких важливе місце займають стекова та купова пам'ять. Ці механізми дозволяють програмам ефективно зберігати дані, такі як змінні, структури та об'єкти, що використовуються під час виконання. Управління динамічною пам'яттю включає процеси виділення та звільнення пам'яті за запитом програмних компонентів. Важливою умовою для забезпечення стабільної роботи є мінімізація фрагментації пам'яті, оскільки накопичення фрагментованих блоків може призводити до неефективного використання ресурсів і погіршення продуктивності.

Часова ефективність будь-якої програми великою мірою залежить від того, наскільки оперативно система обробляє запити на пам'ять, як швидко відбувається виділення необхідних блоків і їхнє подальше звільнення [6]. Фрагментація пам'яті, а також обрані алгоритми доступу до неї, відіграють ключову роль у визначенні загальної продуктивності. Ефективність може значно знижуватися, якщо система не здатна швидко знаходити необхідні ресурси через фрагментовану пам'ять або повільне виконання операцій із виділення та звільнення блоків. На рис. 1.2

продемонстровано основні чинники, які визначають часову ефективність у контексті управління пам'яттю, включаючи фрагментацію, швидкість операцій та особливості доступу до даних.

Рисунок 1.2 – Основні фактори часової ефективності

Часова ефективність методів управління динамічною пам'яттю залежить від таких факторів, як:

– розмір і кількість запитів на виділення пам'яті;

– час, необхідний на виділення та звільнення блоків пам'яті;

– ступінь фрагментації пам'яті;

– спосіб доступу до пам'яті (наприклад, прямий доступ через покажчики чи абстрактні структури) [7].

Найбільш яскравими прикладами використання часової ефективності в управлінні пам'яттю є:

– виділення та звільнення пам'яті. Наприклад, у програмі, що працює з великими масивами даних, необхідно швидко виділяти та ініціалізовувати значні обсяги пам'яті, щоб зберігати ці дані для подальшої обробки. Для цього можуть використовуватися такі функції, як HeapAlloc та VirtualAlloc в ОС Windows. Час, який витрачається на виділення великих блоків пам'яті, безпосередньо впливає на продуктивність програми [8].

– робота з динамічними структурами. Динамічні структури даних, такі як списки або дерева, широко використовуються в програмному забезпеченні для

ефективного зберігання та обробки даних. Важливою складовою є час доступу до елементів цих структур, їх додавання та видалення. Наприклад, використання пов'язаних списків потребує швидкої обробки запитів на додавання та видалення елементів, що є критичним у багатьох програмах реального часу. Часова ефективність операцій із динамічними структурами залежить від алгоритмів, які використовуються для їх реалізації;

– асоціативні структури. Асоціативні структури, такі як словники або хеш-таблиці, використовуються для швидкого доступу до даних за ключем. Важливими аспектами є час на вставку, пошук та видалення елементів. В операційній системі Windows для зберігання таких структур може використовуватися масив пам'яті, керований відповідними алгоритмами пошуку, які мінімізують час доступу до даних. Наприклад, під час зберігання великої кількості об'єктів у словнику критично важливо, щоб операції пошуку ключа та вставки нових значень виконувалися максимально швидко;

– маніпуляції зі стрічками. Стрічки (або рядки) є невід'ємною частиною більшості програм. Часова ефективність операцій із рядками, таких як конкатенація або порівняння, також відіграє важливу роль у продуктивності програмного забезпечення. Наприклад, стандартний оператор конкатенації стрічок може виявитися менш ефективним порівняно зі спеціалізованими класами, такими як StringBuilder, який був розроблений для оптимізації операцій із рядками шляхом мінімізації використання пам'яті та часу виконання [9].

Також варто згадати і фрагментацію, яка виникає, коли вільні блоки пам'яті стають розподіленими між зайнятими блоками, що ускладнює або навіть унеможливлює виділення великих безперервних блоків пам'яті. Це може призводити до збільшення часу на пошук і виділення пам'яті або навіть до помилок у роботі системи через нестачу доступних ресурсів.

Отже, часова ефективність у контексті управління динамічною пам'яттю є критичним фактором, який впливає на продуктивність програмного забезпечення. Ефективне управління пам'яттю дозволяє мінімізувати затримки, знижувати фрагментацію та підвищувати продуктивність системи загалом, що є особливо

важливим для додатків, які працюють у реальному часі або під великими навантаженнями.

1.2 Методи виділення та звільнення пам'яті в ОС Windows

Операційна система Windows забезпечує широкий спектр інструментів для керування пам'яттю. Ці механізми дозволяють програмам ефективно використовувати ресурси системи, виділяти необхідні обсяги пам'яті для виконання завдань та звільняти її, коли вона більше не потрібна. Оскільки пам'ять є обмеженим ресурсом, правильне управління нею є критично важливим для забезпечення стабільної та продуктивної роботи системи.

Windows використовує кілька основних методів для управління пам'яттю, таких як стекова та пам'ять купи, а також механізми для роботи з віртуальною пам'яттю. Кожен із цих методів має свої особливості та застосування в залежності від потреб програмного забезпечення [10].

Стекова пам'ять в Windows використовується для зберігання локальних змінних, параметрів функцій та викликів функцій під час виконання програми. Коли функція викликається, необхідна пам'ять виділяється в стеку, а після завершення роботи функції вона автоматично звільняється. Пам'ять у стеку виділяється та звільняється дуже швидко завдяки його лінійній структурі. Це робить стек ефективним, але обмеженим за розміром ресурсом.

На рис. 1.3 показаний приклад роботи зі стеком – структурою даних, що працює за принципом «останній прийшов – перший пішов» (LIFO). При додаванні нового елемента вказівник стеку зміщується вгору, а при вилученні елементу – знижується до попереднього значення. Стек зручно використовувати для зберігання тимчасових даних під час виконання викликів функцій або обробки рекурсії.

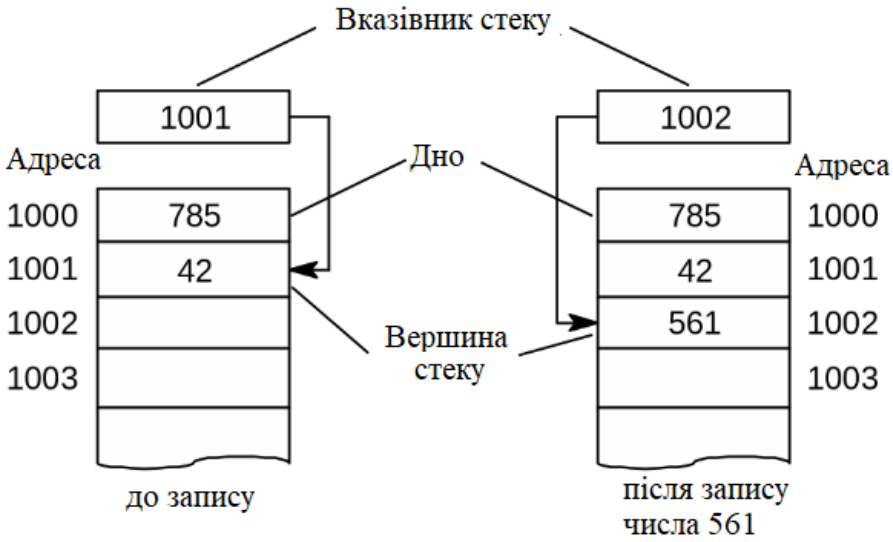

Рисунок 1.3 – Приклад роботи із стеком

Купа (heap) – це область пам'яті, призначена для динамічного розподілу ресурсів під час виконання програми (рис. 1.4). При створенні нових об'єктів у програмі, пам'ять виділяється з купи, причому кожен об'єкт займає безперервний блок пам'яті, що стає недоступним для подальших операцій розподілу. Звільнення пам'яті відбувається, коли об'єкт більше не використовується, і тоді звільнений блок повертається назад у купу, стаючи доступним для нових об'єктів.

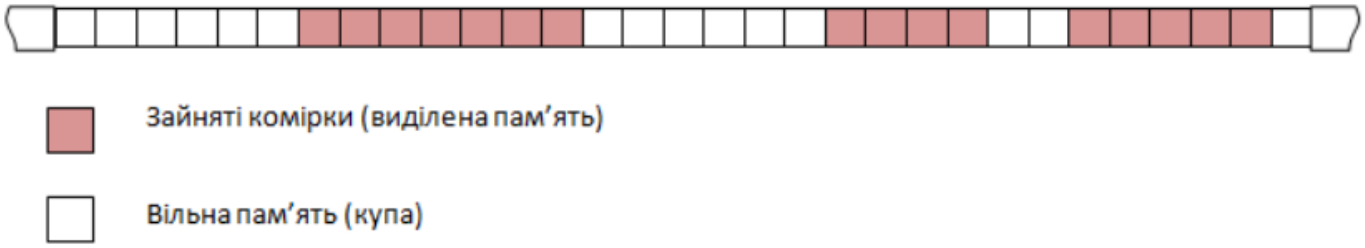

Рисунок 1.4 – Схематичне представлення «купи»

Якщо пам'ять не звільняється або звільняється некоректно, може виникнути витік пам'яті, що призводить до зменшення доступних ресурсів і можливих збоїв у роботі системи. Також можливе явище фрагментації пам'яті, коли виділення великих блоків пам'яті стає неможливим через розрізненість вільних ділянок, навіть якщо загальний обсяг вільної пам'яті достатній. Для управління цим процесом використовується збирач сміття, який автоматично стежить за виділенням і звільненням пам'яті, а також виконує дефрагментацію для оптимізації використання доступних ресурсів.

Пам'ять «купи» не обмежена фіксованими рамками, як стек, і дозволяє програмам виділяти блоки пам'яті довільного розміру, зберігаючи об'єкти та дані протягом усього часу їх використання. Виділення та звільнення пам'яті в купі є більш гнучким процесом порівняно зі стеком, але і більш складним, оскільки не відбувається автоматично – програма повинна явно звільняти пам'ять після її використання [11].

ОС Windows підтримує віртуальну пам'ять, що дозволяє програмам використовувати більше пам'яті, ніж фізично доступно на комп'ютері. Віртуальна пам'ять забезпечує абстракцію фізичної пам'яті і дозволяє операційній системі підмінювати блоки пам'яті, використовуючи сторінки (pages) та сторінковий файл (page file), що зберігається на жорсткому диску. Це значно розширює можливості програм, які можуть працювати з великими обсягами даних.

Для роботи з віртуальною пам'яттю в Windows використовуються функції VirtualAlloc та VirtualFree.

Функція VirtualAlloc в операційній системі Windows використовується для виділення блоків віртуальної пам'яті для програмного забезпечення. Віртуальна пам'ять – це механізм абстракції, який дозволяє програмам отримати доступ до більшого обсягу пам'яті, ніж фізично доступно на комп'ютері. Ця функція дозволяє програмі запросити блок пам'яті конкретного розміру, який може бути набагато більшим за фізичну оперативну пам'ять, оскільки операційна система використовує техніку підкачки (paging). У разі нестачі фізичної пам'яті дані можуть зберігатися в спеціальному файлі підкачки (page file) на жорсткому диску, а операційна система автоматично управляє переміщенням цих даних між оперативною пам'яттю та диском.

При виклику функції VirtualAlloc, програма може також вказати додаткові параметри, такі як бажане розташування блоку пам'яті, доступ до цього блоку (читання, запис) або відкладене виділення, що дозволяє виділяти віртуальну пам'ять без негайного її використання. Це надає гнучкість у плануванні використання ресурсів пам'яті й дозволяє програмам ефективно працювати навіть у випадках інтенсивних обчислень або роботи з великими обсягами даних [12].

Функція VirtualFree використовується для звільнення віртуальної пам'яті, яка була раніше виділена за допомогою VirtualAlloc. Цей процес є критично важливим для підтримання стабільної роботи програм, оскільки без належного звільнення пам'яті може виникнути ситуація, коли пам'ять не буде доступна для інших частин програми чи інших програм, що спричинить витоки пам'яті. VirtualFree повертає віртуальну пам'ять назад операційній системі, дозволяючи їй використовувати цей ресурс для подальших операцій.

Використання VirtualAlloc і VirtualFree є особливо корисним у випадках, коли програма працює з великими обсягами даних або потребує точного контролю за пам'яттю, адже ці функції забезпечують гнучке й ефективне управління віртуальними ресурсами, які є однією з важливих характеристик сучасних операційних систем.

Операційна система Windows забезпечує розробників інструментами для ефективного управління пам'яттю за допомогою спеціалізованих функцій для роботи з купою. Купа (heap) – це область пам'яті, призначена для динамічного виділення блоків під час виконання програм. Для роботи з цією областю пам'яті Windows використовує такі функції, як HeapAlloc та HeapFree, які забезпечують гнучке управління пам'яттю [13].

Функція HeapAlloc дозволяє програмі виділяти блоки пам'яті з купи для зберігання даних під час її виконання. Ключова особливість цієї функції полягає в тому, що операційна система може створювати кілька незалежних куп для оптимізації роботи з пам'яттю. Програма може обрати конкретну купу, з якої буде виділено пам'ять, що особливо корисно в великих або багатопоточних додатках, де необхідно розподіляти навантаження між різними областями пам'яті.

HeapAlloc надає гнучкість у роботі з пам'яттю, дозволяючи виділяти її для різних цілей: зберігання об'єктів, структур даних, буферів для передачі інформації та інших ресурсів, необхідних для виконання програми. Параметри функції можуть визначати такі характеристики, як розмір блоку, який необхідно виділити, та початковий стан пам'яті (наприклад, ініціалізація нулями). Це дозволяє

програмістам краще контролювати процес виділення пам'яті, що знижує ймовірність помилок або витоків пам'яті [14].

Функція HeapFree використовується для звільнення пам'яті, яка була раніше виділена за допомогою HeapAlloc. Після того як програма завершила роботу з певним блоком пам'яті, цей блок потрібно повернути в купу для подальшого використання іншими частинами програми або іншими програмами. Якщо пам'ять не звільняти належним чином, це може призвести до витоків пам'яті, коли ресурси системи вичерпуються, а вільна пам'ять стає недоступною для інших операцій.

Звільнення пам'яті також допомагає уникнути проблеми фрагментації купи. Фрагментація виникає тоді, коли різні блоки пам'яті виділяються і звільняються хаотично, створюючи невеликі розрізнені вільні області, що ускладнює виділення великих блоків пам'яті. Якщо купа буде надмірно фрагментованою, це може погіршити продуктивність програми та навіть призвести до відмови у виділенні пам'яті.

Функції HeapAlloc та HeapFree надають програмістам значний контроль над управлінням пам'яттю, дозволяючи їм ефективно керувати динамічною пам'яттю відповідно до потреб програми. Вони дозволяють виділяти та звільняти пам'ять саме тоді, коли це необхідно, і у потрібному обсязі. Це є важливою перевагою для програм, що працюють з великими обсягами даних або використовують багатопотокову обробку. Однак, використання цих функцій також вимагає уважного підходу до управління пам'яттю, оскільки помилки при звільненні пам'яті можуть призвести до серйозних проблем, таких як витоки пам'яті або фрагментація. Некоректне звільнення пам'яті може призвести до того, що програма буде продовжувати використовувати пам'ять навіть після того, як вона більше не потрібна, що поступово виснажує ресурси системи [15].

Сторінковий обмін (paging) у Windows – це механізм управління пам'яттю, який дозволяє операційній системі переміщувати частини пам'яті (сторінки) між оперативною пам'яттю та диском. Основна мета цього процесу полягає в тому, щоб програми могли використовувати більше пам'яті, ніж фізично доступно, що особливо важливо для роботи з великими обсягами даних. Завдяки підкачуванню

сторінок (paging), операційна система ефективно розподіляє пам'ять, дозволяючи тимчасово зберігати частину даних на диску, а при необхідності завантажувати їх назад у оперативну пам'ять.

На рис. 1.5 зображено приклад роботи сторінкового обміну. Логічна адреса, яку використовує програма, ділиться на номер сторінки та зміщення. Таблиця сторінок (Page Map Table, PMT) зіставляє номер сторінки з фізичною адресою в оперативній пам'яті, де дані зберігаються у відповідних кадрах. Якщо сторінка не знаходиться в оперативній пам'яті, вона завантажується з диска, і процес може продовжити свою роботу.

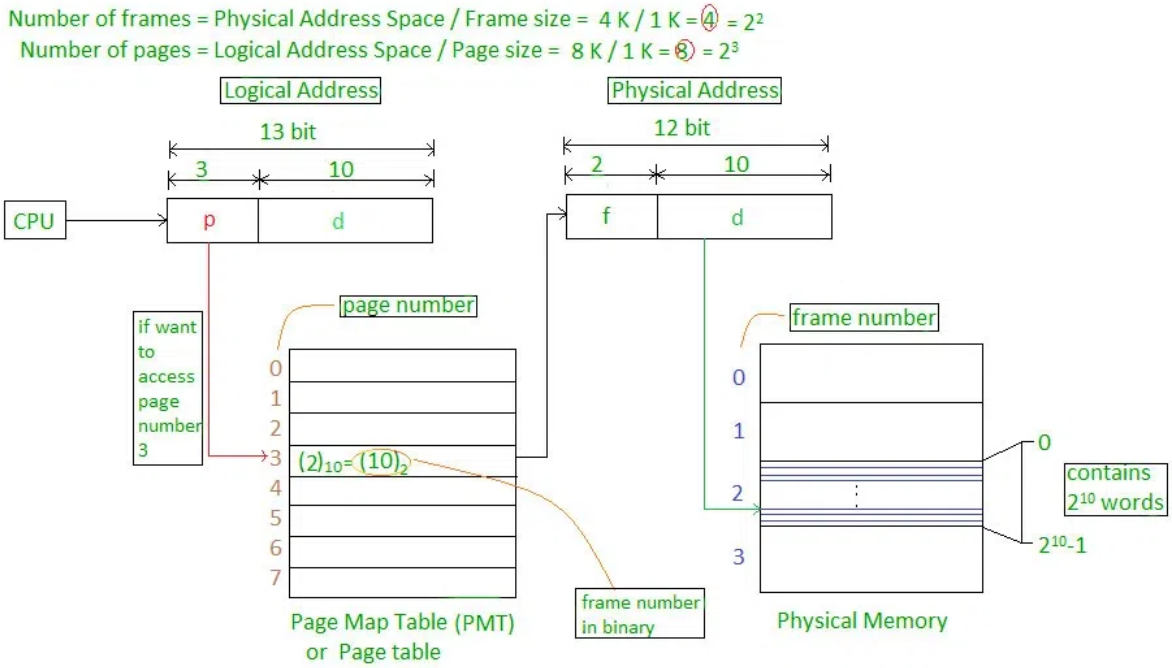

Рисунок 1.5 – Приклад роботи сторінкового обміну [16]

Цей механізм дозволяє програмам використовувати більше пам'яті без перевантаження фізичних ресурсів, забезпечуючи гнучке та ефективне управління пам'яттю.

Операційна система Windows підтримує концепцію пулів пам'яті (memory pools), які призначені для ефективного управління пам'яттю, особливо при роботі з невеликими блоками. Пули пам'яті дозволяють оптимізувати процес виділення і звільнення пам'яті для дрібних об'єктів, що є поширеним у багатьох системних і прикладних задачах. Використання пулів допомагає мінімізувати фрагментацію пам'яті, оскільки невеликі блоки пам'яті виділяються і звільняються швидше і більш

організовано. Це також знижує загальні витрати на обробку запитів на виділення, що підвищує продуктивність системи.

Існує два типи пулів пам'яті: Paged Pool і Non-Paged Pool. Paged Pool – це область пам'яті, яка може бути переміщена до сторінкового файлу на диску, коли операційна система виявляє, що дані з цієї області не використовуються тривалий час. Цей пул використовується для даних, доступ до яких не є критично швидким, тому їх можна тимчасово зберігати на диску, з подальшим поверненням у оперативну пам'ять при необхідності. Такий підхід дозволяє зберегти більше вільної пам'яті для активних процесів, що сприяє загальній ефективності використання пам'яті.

Non-Paged Pool, навпаки, – це область пам'яті, яка завжди залишається в оперативній пам'яті і не може бути переміщена до сторінкового файлу. Ця пам'ять використовується для даних, доступ до яких повинен бути забезпечений без затримок. Це особливо важливо для системних компонентів, таких як драйвери або механізми обробки апаратних переривань, де необхідна висока швидкодія та мінімальна затримка доступу. Такий підхід гарантує, що критично важливі процеси не постраждають через сторінковий обмін, що може викликати затримки при доступі до даних [17].

Таким чином, пули пам'яті в Windows допомагають оптимізувати роботу з невеликими блоками даних і забезпечують більш гнучке управління пам'яттю, розділяючи дані за їхньою важливістю і вимогами до швидкодії. Операційна система Windows надає потужні засоби для управління пам'яттю, які дозволяють ефективно працювати з великими обсягами даних та забезпечують гнучкість програмістам. Використання стекової та пам'яті «купи», механізмів віртуальної пам'яті та сторінкового обміну дозволяє забезпечити високу продуктивність програм, навіть якщо фізична пам'ять обмежена. Кожен із цих методів має свої переваги та обмеження, що робить їх придатними для різних сценаріїв використання в залежності від потреб конкретних програм або систем.

1.3 Огляд існуючих програмних рішень для тестування управління пам'яттю

Сучасні операційні системи та середовища програмування забезпечують автоматизоване управління пам'яттю, однак у багатьох випадках існує необхідність перевірки ефективності цього процесу, виявлення потенційних витоків пам'яті, фрагментації та інших проблем, які можуть знижувати продуктивність програм. Для таких завдань існують спеціальні програмні рішення, що дозволяють здійснювати тестування та аналіз використання пам'яті.

На сьогодні розроблено безліч інструментів для діагностики та оптимізації управління пам'яттю, кожен з яких має свої особливості та функціональні можливості. Ці інструменти варіюються від простих утиліт для моніторингу стану пам'яті до потужних профайлерів, здатних проводити глибокий аналіз використання динамічної пам'яті під час виконання програм. Більшість з них дозволяють виявити проблеми, пов'язані з витоками пам'яті, надмірним використанням ресурсів, фрагментацією та низькою ефективністю використання динамічної пам'яті.

Valgrind – це програмний інструмент, призначений для динамічного аналізу програм, зокрема для виявлення проблем з управлінням пам'яттю та виявлення витоків пам'яті (рис. 1.6). Його основне призначення – допомогти розробникам виявляти помилки в роботі з пам'яттю, такі як неправильне виділення або звільнення, використання некоректних вказівників, а також виявляти інші проблеми, що можуть негативно вплинути на стабільність програми. Valgrind також застосовується для профілювання продуктивності програм, виявлення «узьких місць» і оптимізації їх виконання.

Архітектура Valgrind базується на інструментах, що працюють на рівні симуляції виконання коду. При запуску програми за допомогою Valgrind її виконання відбувається в спеціально створеному середовищі, де відстежуються всі операції з пам'яттю. Valgrind перехоплює інструкції програми, виконує їх у власному віртуальному середовищі й аналізує, як програма взаємодіє з пам'яттю. Завдяки такій архітектурі Valgrind може виявляти низькорівневі помилки, які важко виявити стандартними засобами тестування. Окрім цього, Valgrind дозволяє запускати програми без необхідності змінювати вихідний код, що спрощує

інтеграцію цього інструмента в процес розробки та тестування програмного забезпечення.

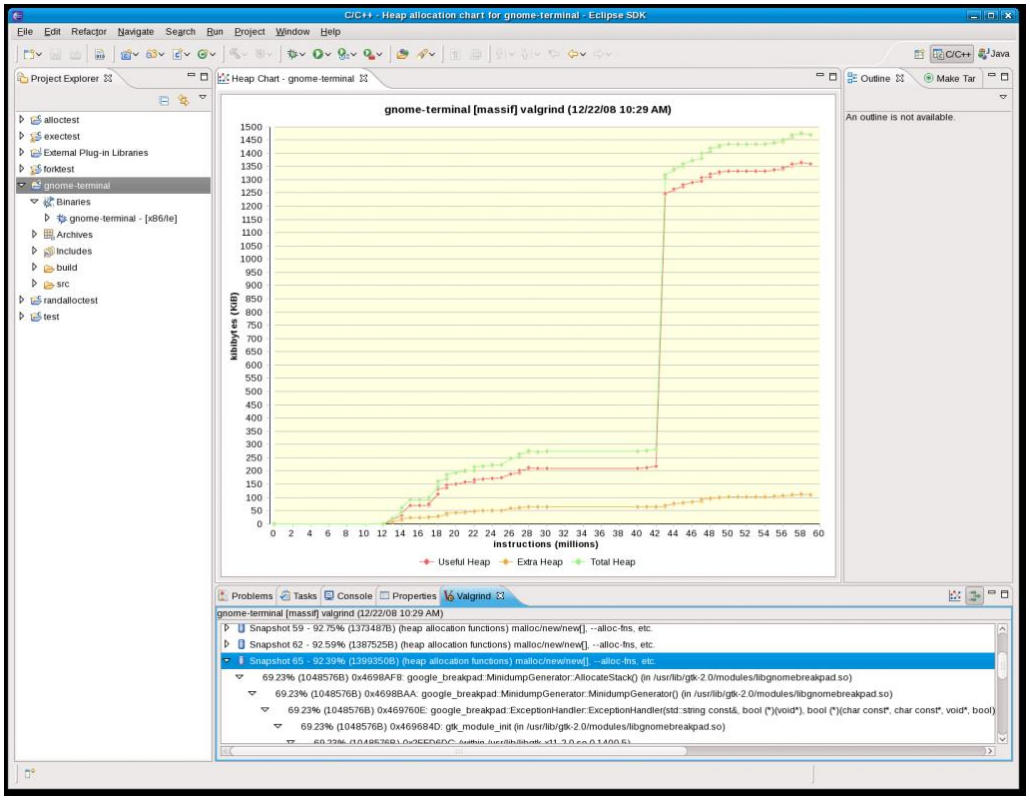

Рисунок 1.6 – Приклад інтерфейсу системи «Valgrind» [18]

Valgrind підтримує різні інструменти для аналізу пам'яті, включаючи Memcheck, який є основним модулем для виявлення витоків пам'яті та некоректних операцій. Завдяки своїй гнучкій архітектурі, Valgrind може бути розширений за допомогою додаткових модулів, що дозволяє використовувати його для вирішення різних задач, від виявлення помилок в управлінні пам'яттю до профілювання продуктивності програм.

Переваги Valgrind:

– виявлення помилок в управлінні пам'яттю. Valgrind ефективно виявляє витоки пам'яті, використання невизначених змінних, неправильне звільнення пам'яті та інші помилки, які важко діагностувати вручну;

– модульна архітектура. Завдяки підтримці кількох інструментів, таких як Memcheck, Callgrind та інші, Valgrind дозволяє адаптувати функціональність під різні потреби, від аналізу пам'яті до профілювання продуктивності;

– підтримка різних мов програмування. Valgrind можна використовувати з програмами, написаними на мовах C, C++, Fortran та інших, що робить його універсальним інструментом;

– працює без модифікації коду. Програми можна запускати через Valgrind без необхідності вносити зміни у вихідний код, що значно спрощує процес інтеграції.

Недоліки Valgrind:

– зниження продуктивності. Використання Valgrind значно уповільнює виконання програм, оскільки вони запускаються у віртуальному середовищі для аналізу;

– обмежена підтримка платформ. Valgrind не підтримується на всіх операційних системах, зокрема, його не можна використовувати на Windows без сторонніх модифікацій;

– не виявляє всі типи помилок. Valgrind в основному сфокусований на помилках, пов'язаних із пам'яттю, але може пропустити інші проблеми продуктивності або баги, які не пов'язані з управлінням пам'яттю;

– високі вимоги до ресурсів. Через роботу в симуляційному середовищі Valgrind потребує більше ресурсів системи, що може бути проблемою для великих програм або систем із обмеженими можливостями [19].

Отже, Valgrind – це потужний інструмент для динамічного аналізу програм і виявлення помилок в управлінні пам'яттю. Його основні переваги включають модульну архітектуру, виявлення витоків пам'яті та сумісність із різними мовами програмування. Однак через зниження продуктивності, обмежену підтримку платформ та високі вимоги до ресурсів, Valgrind може бути не найкращим вибором для деяких типів програм або великих проектів. Незважаючи на це, він залишається одним із найкращих інструментів для глибокого аналізу пам'яті в програмах, написаних на мовах C і C++.

HeapMemView – це невелика утиліта для Windows, призначена для відстеження та аналізу використання пам'яті «купи» програмами, які працюють у системі (рис. 1.7). Основне призначення цієї утиліти полягає в тому, щоб показувати

розробникам та системним адміністраторам інформацію про блоки пам'яті, які були виділені і використовуються додатками у купі. Цей інструмент є корисним для діагностики проблем із пам'яттю, таких як витоки пам'яті або фрагментація.

Архітектура HeapMemView побудована таким чином, щоб зчитувати інформацію безпосередньо з процесів, які працюють у системі. Утиліта не потребує жодних змін у вихідному коді програм, а натомість взаємодіє безпосередньо з операційною системою, щоб отримати детальні дані про виділені блоки пам'яті в купі. HeapMemView дозволяє бачити розміри блоків, адреси, статуси виділення, а також деталі про те, яка програма використовує конкретні ділянки пам'яті. Оскільки цей інструмент працює на рівні операційної системи, він може виявляти проблеми, пов'язані з пам'яттю, без необхідності втручання в роботу додатків.

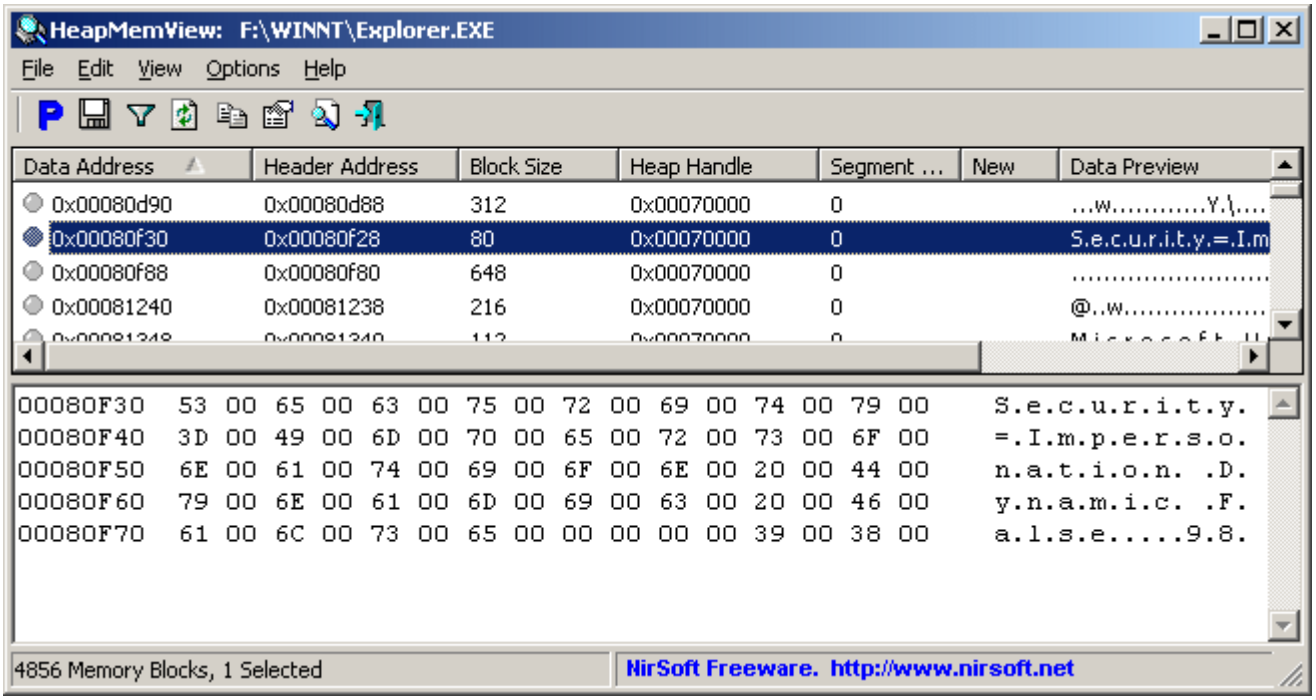

Рисунок 1.7 – Приклад інтерфейсу системи «HeapMemView» [20]

Переваги HeapMemView:

– простота використання. Утиліта має простий і зрозумілий інтерфейс, що дозволяє користувачам швидко отримати інформацію про використання пам'яті без складних налаштувань;

– не потребує змін у коді. HeapMemView дозволяє аналізувати використання пам'яті в додатках без необхідності модифікувати їх вихідний код;

– легка і швидка. Утиліта є дуже легкою, споживає мінімальні системні ресурси і швидко виконує аналіз, що робить її зручною для використання в режимі реального часу;

– безкоштовна. HeapMemView доступна безкоштовно, що робить її доступною для широкого кола користувачів.

Недоліки HeapMemView:

– обмежена функціональність. Утиліта має досить вузьку функціональність, надаючи лише базову інформацію про купову пам'ять, що може не відповідати потребам складного аналізу;

– немає підтримки інших типів пам'яті. HeapMemView фокусується виключно на куповій пам'яті, не надаючи інформації про стекову або віртуальну пам'ять;

– відсутність глибокого аналізу. Утиліта не дозволяє проводити глибокий аналіз або профілювання продуктивності, як це роблять складніші інструменти;

– обмежена підтримка великих проектів. Для великих або багатопотокових програм HeapMemView може бути недостатньо потужною, оскільки вона не призначена для глибокого аналізу складних систем.

Отже, HeapMemView – це корисна утиліта для швидкого і легкого аналізу використання купової пам'яті в Windows. Вона відзначається простотою використання та невеликими системними вимогами, що робить її ідеальною для базової діагностики проблем із пам'яттю. Однак її обмежена функціональність та відсутність можливостей для глибокого аналізу роблять її менш придатною для складних або великих проектів, що вимагають детальнішого підходу до управління пам'яттю.

PerfMon (Performance Monitor) – це вбудований інструмент Windows, призначений для моніторингу продуктивності системи в режимі реального часу та збору даних про різні компоненти системи, такі як використання процесора, пам'яті, дискових операцій та мережевої активності (рис. 1.8). Основна мета PerfMon – допомогти системним адміністраторам та розробникам аналізувати продуктивність системи, виявляти вузькі місця в роботі програм або системних ресурсів, а також

діагностувати проблеми, що впливають на стабільність і ефективність роботи операційної системи.

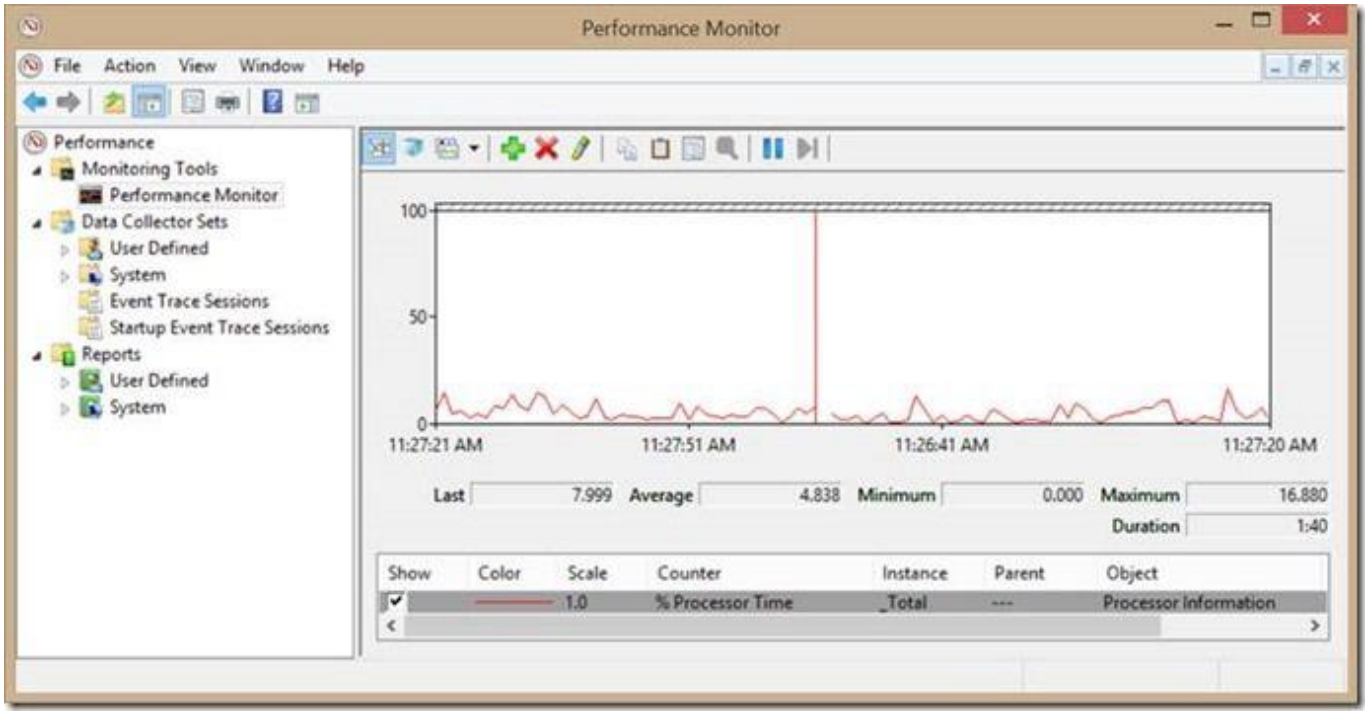

Рисунок 1.8 – Приклад інтерфейсу системи «PerfMon» [21]

Архітектура PerfMon базується на концепції лічильників продуктивності (performance counters), які збирають інформацію про різні компоненти системи. Кожен лічильник представляє окремий параметр, такий як завантаження процесора, кількість використовуваної пам'яті або активність диска. PerfMon надає можливість вибору з великої кількості цих лічильників для створення спеціальних наборів даних, що можуть бути зібрані та збережені для подальшого аналізу. За допомогою цього інструменту можна як переглядати продуктивність у режимі реального часу, так і налаштовувати тривалі моніторингові сесії для збирання статистики за певний період.

Переваги PerfMon:

– вбудований інструмент Windows. PerfMon не вимагає додаткової установки або стороннього ПЗ, оскільки він інтегрований у систему і готовий до використання;

– гнучкий моніторинг. Користувачі можуть налаштовувати різноманітні лічильники для моніторингу практично всіх системних компонентів, що дозволяє адаптувати його під конкретні потреби;

– підтримка оповіщень. PerfMon може автоматично генерувати попередження при досягненні критичних показників, що допомагає вчасно реагувати на проблеми з продуктивністю;

– збір даних у реальному часі та тривалий моніторинг. Інструмент дозволяє як спостерігати за продуктивністю системи в реальному часі, так і проводити тривалі сеанси збору даних для подальшого аналізу.

Недоліки PerfMon:

– складний інтерфейс для новачків. Попри свою гнучкість, інтерфейс PerfMon може бути складним для користувачів, які не мають досвіду в налаштуванні системних лічильників і аналізі продуктивності;

– обмежені можливості аналізу. PerfMon більше підходить для збору даних, але не для глибокого аналізу. Для більш детальної діагностики можуть знадобитися інші інструменти;

– висока залежність від налаштувань користувача. Для отримання точних і корисних даних користувачі повинні самостійно налаштовувати лічильники і фільтри, що може бути складним завданням;

– не працює поза Windows. Інструмент призначений виключно для Windows, що обмежує його використання в багатоплатформних середовищах.

Отже, PerfMon є потужним інструментом для моніторингу продуктивності в операційній системі Windows. Його інтеграція в систему, гнучкі налаштування та можливість тривалого моніторингу роблять його ефективним засобом для відстеження стану ресурсів і системних компонентів. Однак складний інтерфейс і обмежені можливості аналізу можуть вимагати додаткових знань або використання сторонніх інструментів для більш детальної діагностики. Незважаючи на це, PerfMon залишається основним вибором для базового моніторингу продуктивності на платформі Windows [22].

Враховуючи проведений аналіз існуючих програмних рішень для тестування управління пам'яттю, можна зробити висновок про необхідність розробки нового інструменту, який надасть можливість більш детально аналізувати складні операції з пам'яттю. Сучасні рішення або мають обмежену функціональність, або не

забезпечують достатньо глибокого аналізу динамічних структур, великих масивів та асоціативних структур із врахуванням часу виконання, використання пам'яті та навантаження на процесор. Запропонована система заповнить цю нішу, забезпечуючи вимірювання продуктивності при роботі з великими масивами, динамічними структурами і складними операціями з пам'яттю, а також дозволить аналізувати ефективність управління ресурсами у реальних умовах фрагментації пам'яті.

1.4 Постановка задачі

Розробка нової системи для тестування та аналізу управління пам'яттю в операційній системі Windows спрямована на вирішення завдань моніторингу та оптимізації процесів виділення, звільнення та використання пам'яті. Система повинна надати користувачам можливість досліджувати різні сценарії роботи з пам'яттю, виконуючи експерименти з великими масивами даних, динамічними структурами, такими як списки, словники, а також фрагментованою пам'яттю. Основною метою є виявлення та аналіз впливу цих операцій на продуктивність системи з метою її подальшої оптимізації. Система повинна підтримувати можливість задавати кількість експериментів для кожного сценарію, що дозволить детально досліджувати поведінку пам'яті за різних умов.

Кожен експеримент має бути орієнтований на точне вимірювання ключових параметрів роботи з пам'яттю. Це включає:

– час виконання кожної операції – від моменту початку виділення пам'яті до її звільнення та завершення операції;

– обсяг використаної пам'яті, що дасть змогу оцінити ефективність операцій з пам'яттю, особливо при роботі з великими масивами чи фрагментованою пам'яттю;

– середнє навантаження на процесор під час виконання кожного експерименту – важливий показник, що впливає на загальну продуктивність системи;

– доступну пам'ять перед початком і після завершення операцій – це допоможе виявити проблеми з витоками пам'яті та неефективне управління ресурсами.

Для кожного сценарію система повинна дозволяти створювати множинні експерименти з різними вхідними параметрами. Наприклад, при роботі з фрагментованою пам'яттю користувач може задавати різні розміри фрагментів та кількість блоків для кожного експерименту. Це дозволить провести серію тестів з різними умовами та виявити, як фрагментація впливає на час доступу до пам'яті, швидкість виділення та звільнення блоків, а також на загальне навантаження на процесор. Важливо забезпечити підтримку примусової збірки сміття, що дозволить створювати сценарії з різним рівнем фрагментації та оцінювати її вплив на роботу системи.

Одним із важливих аспектів є візуалізація отриманих результатів. Система повинна надавати зручні інструменти для графічного відображення даних про час виконання та використання пам'яті кожного експерименту. Візуалізація допоможе користувачам швидко і зручно аналізувати результати, порівнювати різні стратегії та оцінювати вплив різних параметрів на продуктивність системи. Завдяки цьому користувач зможе визначити найефективніші підходи до управління пам'яттю, вибираючи оптимальні стратегії для своїх завдань.

Система повинна бути гнучкою у налаштуваннях, що дозволить легко модифікувати параметри кожного експерименту, змінюючи кількість фрагментів, розмір блоків пам'яті, типи структур даних і алгоритми. Це дозволить користувачам адаптувати її під різні сценарії роботи з пам'яттю, що є важливим для тестування програмного забезпечення в умовах реальних навантажень.

2 ВИБІР ІНСТРУМЕНТІВ ТА ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Формулювання вимог до системи для дослідження часової ефективності

Розробка системи для дослідження часової ефективності управління пам'яттю вимагає визначення чітких функціональних вимог, які забезпечать необхідну гнучкість та ефективність для проведення різних експериментів із пам'яттю [23]. Основне завдання системи полягає у вимірюванні та візуалізації часу виконання операцій з пам'яттю, використання ресурсів та продуктивності при роботі з динамічними структурами, масивами, пов'язаними списками, словниками та іншими структурами даних.

У табл. 2.1 наведено основні функціональні вимоги до розробки системи.

Таблиця 2.1 – Функціональні вимоги до системи

Вимога	Опис	Параметри вимірювання	Результат/Вивід
1	2	3	4
Виділення та ініціалізація масивів	Виділення великих масивів у пам'яті з подальшою ініціалізацією. Користувач задає розмір масиву та кількість операцій.	Час виконання, використання пам'яті, навантаження на процесор	Графічна візуалізація часу виконання та використання пам'яті для кожного експерименту
Операції з динамічними структурами	Проведення операцій додавання, видалення та доступу до елементів у динамічних структурах, таких як списки.	Час виділення, використання пам'яті, навантаження на процесор	Порівняльний аналіз продуктивності при різних розмірах структур
Операції з пов'язаними списками	Додавання, видалення та обхід елементів у пов'язаних списках з вимірюванням часу та використання пам'яті для кожного експерименту.	Час виконання, використання пам'яті, навантаження на процесор	Графічний аналіз ефективності роботи зі списками при різних розмірах списку

Продовження таблиці 2.1.

1	2	3	4
Робота зі словниками	Тестування операцій додавання, доступ до елементів та видалення елементів у словниках з вимірюванням продуктивності та пам'яті.	Час виконання, використання пам'яті, навантаження на процесор	Графічна візуалізація часу операцій та продуктивності роботи зі словниками
Маніпуляції зі стрічками	Генерація кількох варіацій початкового рядка через циклічну зміну символів ASCII для кожного символу з оцінкою продуктивності.	Час виділення, використання пам'яті, навантаження на процесор	Порівняння часу генерації рядків, оцінка ефективності маніпуляцій через стандартні методи та спеціалізовані алгоритми
Виділення на великій купі об'єктів	Виділення великих масивів байтів на купі, з подальшою ініціалізацією кожного масиву байтів та їх збереженням у списку для подальшого використання.	Час виконання, використання пам'яті, навантаження на процесор	Оцінка часу виділення пам'яті та продуктивності роботи з великими масивами, аналіз використання системних ресурсів
Фрагментація пам'яті	Виділення блоків різного розміру в пам'яті з подальшим випадковим звільненням частини блоків для створення фрагментації. Примусова збірка сміття для очищення пам'яті.	Час виконання, використання пам'яті, навантаження на процесор	Аналіз впливу фрагментації на ефективність управління пам'яттю, вимірювання часу виділення та звільнення блоків пам'яті
Небезпечний код і покажчики	Робота з небезпечним кодом і покажчиками для тестування низькорівневих операцій із пам'яттю, включаючи доступ до елементів через покажчики.	Час виконання, використання пам'яті, навантаження на процесор	Оцінка ефективності низькорівневих операцій, порівняння продуктивності між кількістю операцій для роботи із покажчиками

Для кожної операції, яку тестуватиме користувач, система повинна надавати графіки, що відображають час виконання операції та обсяг використаної пам'яті. Це

дозволить користувачам легко порівнювати ефективність різних алгоритмів і сценаріїв роботи з пам'яттю, а також виявляти потенційні проблеми з управлінням пам'яттю, такі як витоки пам'яті або надмірне використання ресурсів. Система повинна бути гнучкою та налаштовуваною, з можливістю змінювати параметри кожного експерименту, надавати детальні вимірювання для різних типів структур даних і сценаріїв використання пам'яті, а також візуалізувати отримані результати для подальшого аналізу та оптимізації.

Нефункціональні вимоги є важливими характеристиками системи, які не стосуються безпосередньо її функціональності, але забезпечують зручність використання, стабільність та продуктивність. Вони впливають на загальну якість системи, її масштабованість, продуктивність, надійність та інші параметри, що забезпечують успішне виконання функціональних вимог. Для розроблюваної системи, яка аналізуватиме часову ефективність роботи з пам'яттю, сформульовано нефункціональні вимоги та приведено у табл. 2.2.

Таблиця 2.2 – Нефункціональні вимоги до системи

Вимога	Опис	Очікуваний результат
1	2	3
Продуктивність	Система повинна мінімізувати власне навантаження на ресурси під час проведення експериментів, щоб результати були максимально точними.	Мінімальний вплив на час виконання та використання пам'яті тестованих програм.
Масштабованість	Система повинна коректно працювати як з малими, так і з великими наборами даних, забезпечуючи стабільну роботу незалежно від обсягу інформації.	Стабільна продуктивність незалежно від обсягу тестованої пам'яті або кількості операцій.
Надійність	Система повинна обробляти виняткові ситуації без збоїв, включаючи некоректні операції з пам'яттю, проблеми з доступом та помилки під час виконання.	Захист від збоїв, коректне завершення роботи в разі виняткових ситуацій, збереження результатів тестування.

Продовження таблиці 2.2.

1	2	3
Зручність використання	Інтерфейс системи має бути інтуїтивно зрозумілим і зручним для налаштування експериментів та аналізу результатів.	Легкий доступ до налаштувань експериментів, інтуїтивний інтерфейс для перегляду графіків та таблиць з результатами.
Безпека	Підтримка виконання небезпечних операцій з пам'яттю повинна бути захищена від помилок, які можуть негативно вплинути на роботу всієї системи.	Безпечне виконання небезпечного коду, ізоляція критичних операцій для уникнення негативного впливу на систему.
Точність вимірювань	Система повинна забезпечувати високоточні вимірювання часу виконання, використання пам'яті та навантаження на процесор.	Точне вимірювання параметрів без відхилень, забезпечення високої роздільної здатності вимірювань.
Сумісність	Система повинна підтримувати різні версії Windows та архітектури (x86, x64).	Коректна робота на різних платформах та підтримка різних середовищ для тестування програм з пам'яттю.
Модульність і розширюваність	Система повинна мати модульну архітектуру, що дозволить легко додавати нові функції або адаптувати існуючі під нові потреби користувачів.	Можливість додавання нових модулів для тестування або візуалізації без потреби в суттєвих змінах архітектури системи.
Моніторинг ресурсів	Система повинна відстежувати використання системних ресурсів, таких як пам'ять і процесор, під час експериментів, без створення надмірного навантаження.	Реальне відображення використання ресурсів у процесі тестування без значного впливу на саму систему, збалансоване навантаження на ресурси під час моніторингу.

Нефункціональні вимоги у свою чергу забезпечать створення стабільної, гнучкої та масштабованої системи, яка здатна проводити точний аналіз часової ефективності управління пам'яттю у різних сценаріях. Висока продуктивність, надійність, точність вимірювань та зручний інтерфейс – основні критерії, які

дозволять користувачам ефективно використовувати систему для тестування та оптимізації програмного забезпечення.

2.2 Проєктування діаграм основних процесів

Проєктування системи для дослідження часової ефективності управління пам'яттю вимагає чіткого розуміння основних процесів, які вона повинна виконувати. Основною метою проєктування є створення чіткого бачення того, як система взаємодіятиме з користувачем та яким чином будуть організовані внутрішні операції для вимірювання часу виконання, використання пам'яті та навантаження на процесор. Процес проєктування охоплює всі аспекти роботи з пам'яттю: від виділення масивів до роботи з небезпечним кодом, що дозволить забезпечити системі повноцінну функціональність для тестування різних сценаріїв.

На початковому етапі важливо виділити основні сценарії використання системи, які включають такі операції, як виділення та ініціалізація масивів, операції з динамічними структурами, робота з пов'язаними списками, словниками та інші. Кожен із цих процесів буде мати свої особливості, і кожен експеримент буде вимагати різних вхідних параметрів, які налаштовуються користувачем перед запуском.

Однією з ключових частин проєктування є діаграми основних процесів, які дозволяють графічно відобразити, як система повинна працювати в різних сценаріях. Діаграми варіантів використання (use case diagrams) показують, які операції будуть доступні користувачу та як вони взаємодіятимуть із системою для виконання експериментів [24]. Це дозволяє візуалізувати основні ролі, які відіграватимуть користувачі, та кроки, які потрібно буде виконати для запуску різних тестів пам'яті.

Діаграма варіантів використання також служить основою для подальшої деталізації кожного сценарію. Нижче подано табл. 2.3, у якій зроблено опис основних прецедентів використання системи.

Таблиця 2.3 – Варіанти використання системи

№	Назва прецеденту	Опис
1	Виділяти та ініціалізувати масиви	Процес виділення пам'яті для великих масивів і їх ініціалізацію. Користувач задає розмір масиву, після чого система вимірює час виділення пам'яті та ініціалізації, а також використання ресурсів.
2	Виконувати операції з динамічними структурами	Робота з динамічними структурами, такими як списки. Система повинна оцінювати ефективність додавання, видалення та доступу до елементів у цих структурах з вимірюванням часу й пам'яті.
3	Виконувати операції з пов'язаними списками	Прецедент включає операції додавання, видалення та обходу елементів у пов'язаних списках. Система вимірює час для кожної операції та ресурсні витрати, такі як пам'ять і навантаження на процесор.
4	Працювати зі словниками	Робота зі словниками, де буде додавання елементів у словник та операції з елементами словника (доступ до елементів і видалення). Система оцінює ефективність цих операцій з вимірюванням часу й використаної пам'яті.
5	Виконувати маніпуляції зі стрічками	Прецедент включає операції з маніпуляціями зі строковими даними, такими як конкатенація, копіювання та шифрування рядків. Система вимірює час виконання операцій та витрати пам'яті під час роботи з рядками.
6	Виділяти пам'ять на великій купі об'єктів	Прецедент стосується виділення пам'яті для великих об'єктів у купі. Система здійснює вимірювання часу виділення та ініціалізації кожного об'єкта, а також аналізує вплив на системні ресурси.
7	Створювати фрагментацію пам'яті	Прецедент описує процес створення фрагментованої пам'яті шляхом виділення блоків різного розміру і їх подальшого звільнення. Система вимірює ефективність операцій у фрагментованій пам'яті та роботу збирача сміття.
8	Виконувати операції з небезпечним кодом та покажчиками	Прецедент включає операції з небезпечним кодом та покажчиками, дозволяючи користувачу працювати безпосередньо з пам'яттю. Система вимірює час доступу до елементів через покажчики та аналізує вплив на продуктивність.

На основі визначених прецедентів була побудована діаграма варіантів використання системи, яка відображає основні дії, доступні користувачу (рис. 2.1).

Ця діаграма ілюструє, як користувач взаємодіє із системою для проведення різних операцій з пам'яттю, включаючи виділення масивів, роботу з динамічними структурами, пов'язаними списками, словниками, маніпуляціями зі стрічками та інші дії. Кожен із цих варіантів використання демонструє конкретні можливості системи для оцінки ефективності управління пам'яттю в ОС Windows.

Рисунок 2.1 – Діаграма варіантів використання системи

На рис. 2.2 представлена діаграма класів, яка відображає структуру для зберігання результатів експериментів із дослідження часової ефективності роботи з пам'яттю. Діаграма показує зв'язок між основним класом для збереження результатів і похідними класами, які реалізують різні типи операцій із пам'яттю, такими як робота зі словниками, динамічними структурами, небезпечним кодом і покажчиками, а також маніпуляціями зі стрічками. Кожен із цих класів зберігає результати експериментів, включаючи час виконання, використання пам'яті та середнє навантаження на процесор.

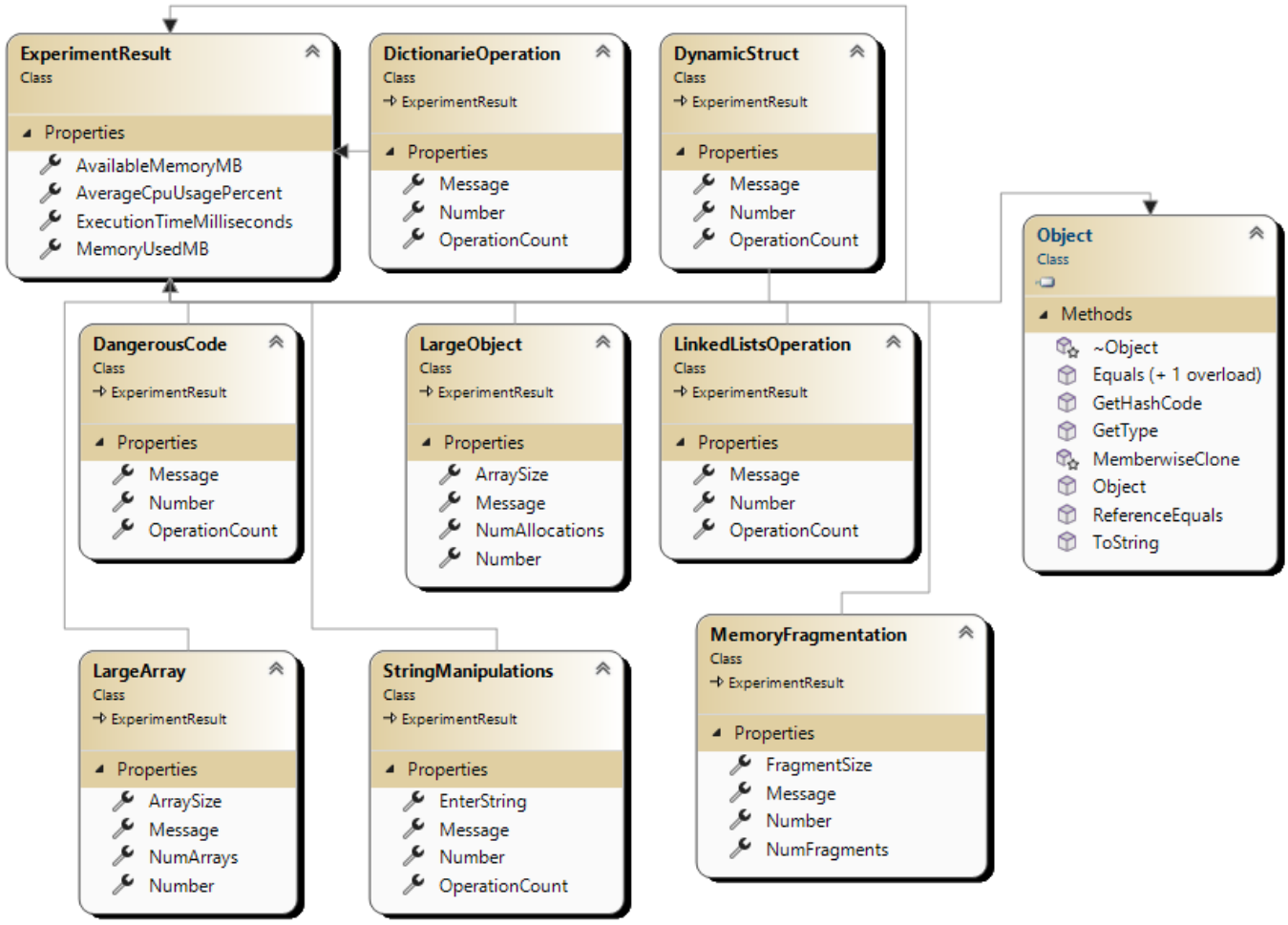

Рисунок 2.2 – Діаграма класів для маніпуляції із даними системи

Діаграма класів складається із:

– ExperimentResult – основний клас, який зберігає результати експерименту. Містить такі властивості, як: час виконання (ExecutionTimeMilliseconds), використана пам'ять (MemoryUsedMB), середнє навантаження на процесор (AverageCpuUsagePercent) та доступна пам'ять (AvailableMemoryMB);

– DangerousCode – клас для збереження результатів експериментів з небезпечним кодом і покажчиками. Містить: Number (Номер експерименту), OperationCount (Кількість операцій) та Message (Повідомлення про результат).

– DictionarieOperation – клас для роботи зі словниками, зберігає результати операцій додавання, видалення та пошуку. Містить поля: Number (Номер експерименту), OperationCount (Кількість операцій) та Message (Повідомлення про результат);

– DynamicStruct – клас для роботи з динамічними структурами, такими як списки та черги. Містить наступні поля: Number (Номер експерименту), OperationCount (Кількість операцій) та Message (Повідомлення про результат);

– LargeArray – клас для виділення та ініціалізації масивів. Містить: Number (Номер експерименту), NumArrays (Кількість масивів), ArraySize (Розмір масиву) та Message (Повідомлення про результат);

– LargeObject – клас для виділення пам'яті на великій купі об'єктів. Містить поля: Number (Номер експерименту), NumAllocations (Кількість виділених об'єктів),

– ArraySize (Розмір масивів) та Message (Повідомлення про результат);

– LinkedListsOperation – клас для операцій із пов'язаними списками. Містить: Number (Номер експерименту), OperationCount (Кількість операцій) та Message (Повідомлення про результат);

– MemoryFragmentation – клас для експериментів із фрагментацією пам'яті. Містить: Number (Номер експерименту), NumFragments (Кількість фрагментів), FragmentSize (Розмір фрагменту) та Message (Повідомлення про результат);

– StringManipulations – клас для маніпуляцій зі стрічками, таких як конкатенація, копіювання та шифрування рядків. Містить наступні поля: Number (Номер експерименту), OperationCount (Кількість операцій), EnterString (Початковий рядок) та Message (Повідомлення про результат).

Проєктування інтерфейсу системи спрямоване на забезпечення зручності використання для кінцевих користувачів. Інтерфейс повинен бути інтуїтивно зрозумілим і дозволяти користувачам легко налаштовувати параметри експериментів, запускати їх та переглядати результати. Особлива увага приділяється візуалізації отриманих даних, що забезпечує чітке уявлення про часову ефективність операцій з пам'яттю, використання ресурсів та навантаження на процесор. Інтерфейс також дозволяє користувачам зберігати та аналізувати результати експериментів для подальшого дослідження.

На рис. 2.3 представлена діаграма класів інтерфейсу системи, яка відображає основні елементи, що забезпечують взаємодію користувача із системою.

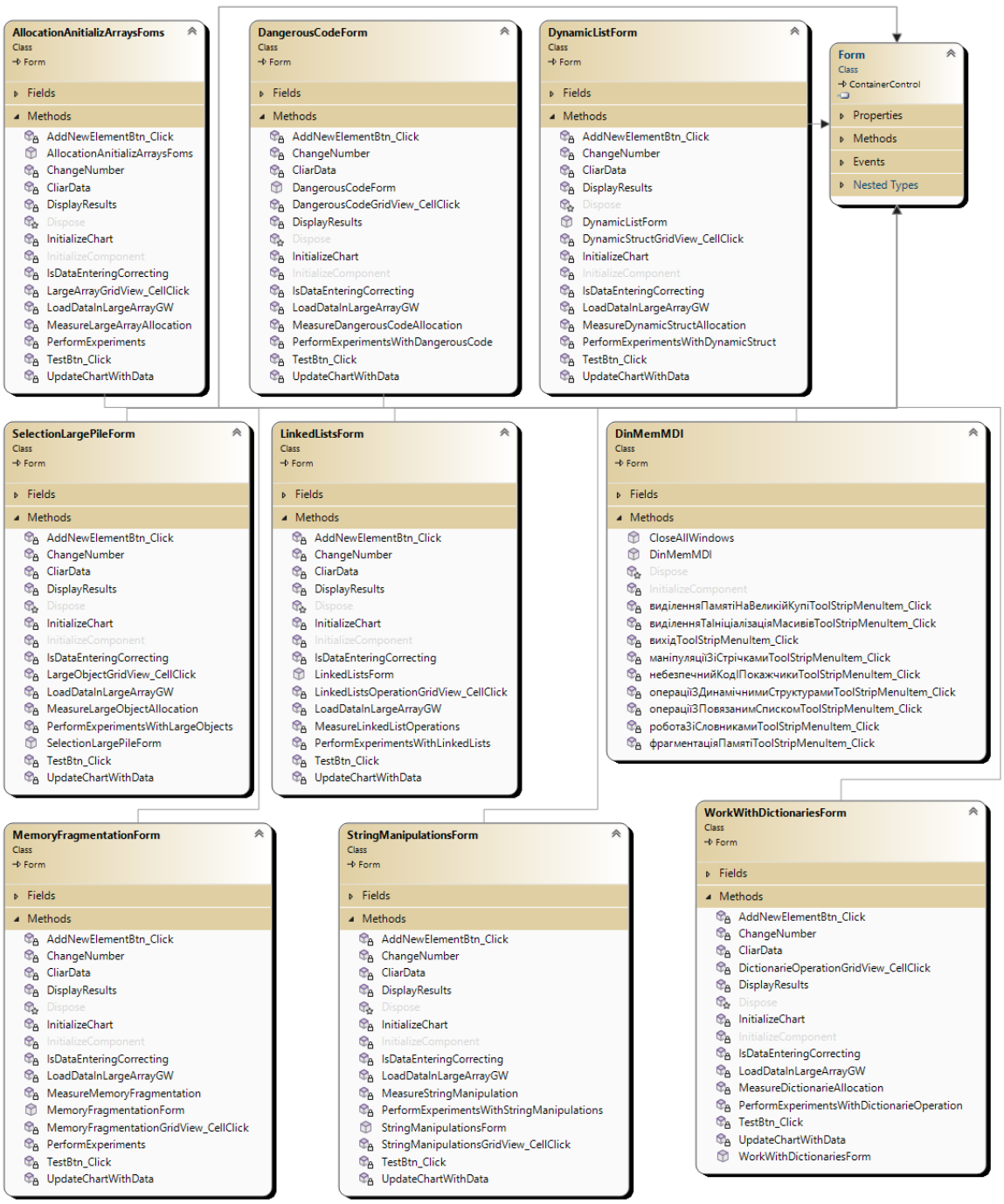

Рисунок 2.3 – Діаграма класів інтерфейсу системи

На діаграмі представлено класи, які дозволяють користувачеві проводити різноманітні експерименти з управління пам'яттю, збирати результати та аналізувати їх для подальшої оптимізації системи:

– DinMemMDI – головне вікно із меню. Це основний клас, який відповідає за навігацію в системі та вибір конкретних типів експериментів для запуску;

– AllocationAnitializArraysForm – форма для дослідження ефективності виділення та ініціалізації масивів. Забезпечує налаштування параметрів для тестування масивів та збору результатів;

– DangerousCodeForm – форма для дослідження ефективності роботи з небезпечним кодом і покажчиками. Користувач може задавати параметри для тестування низькорівневих операцій з пам'яттю;

– DynamicListForm – форма для дослідження ефективності операцій з динамічними структурами, такими як списки, стеки та черги. Вона дозволяє налаштовувати і запускати експерименти з різними типами динамічних структур;

– LinkedListsForm – форма для дослідження ефективності операцій з пов'язаним списком. Користувач може оцінити швидкість додавання, видалення та обходу елементів у списках;

– MemoryFragmentationForm – форма для дослідження ефективності при фрагментації пам'яті. Забезпечує симуляцію фрагментації та вимірювання впливу на продуктивність системи;

– SelectionLargePileForm – форма для дослідження ефективності виділення на великій купі об'єктів. Вона дозволяє користувачеві визначати кількість та розмір об'єктів для тестування виділення пам'яті;

– StringManipulationsForm – форма для дослідження ефективності маніпуляцій зі стрічками. Включає експерименти з конкатенацією, копіюванням і шифруванням рядків.

– WorkWithDictionariesForm – форма для дослідження ефективності роботи зі словниками (асоціативними масивами). Оцінює операції додавання, видалення та пошуку в словниках.

Діаграми активності використовуються для відображення послідовності дій у процесах або алгоритмах, що виконуються в системі. Вони показують, як системні операції змінюють стани і взаємодіють з користувачем або іншими компонентами. У контексті системи для дослідження часової ефективності управління пам'яттю, діаграми активності дозволяють відобразити, як виконується тестування та зберігаються результати експериментів.

На рис. 2.4 представлена діаграма активності процесу вимірювання параметрів ефективності із операціями пов’язаними із виділенням пам’яті та ініціалізацією масивів.

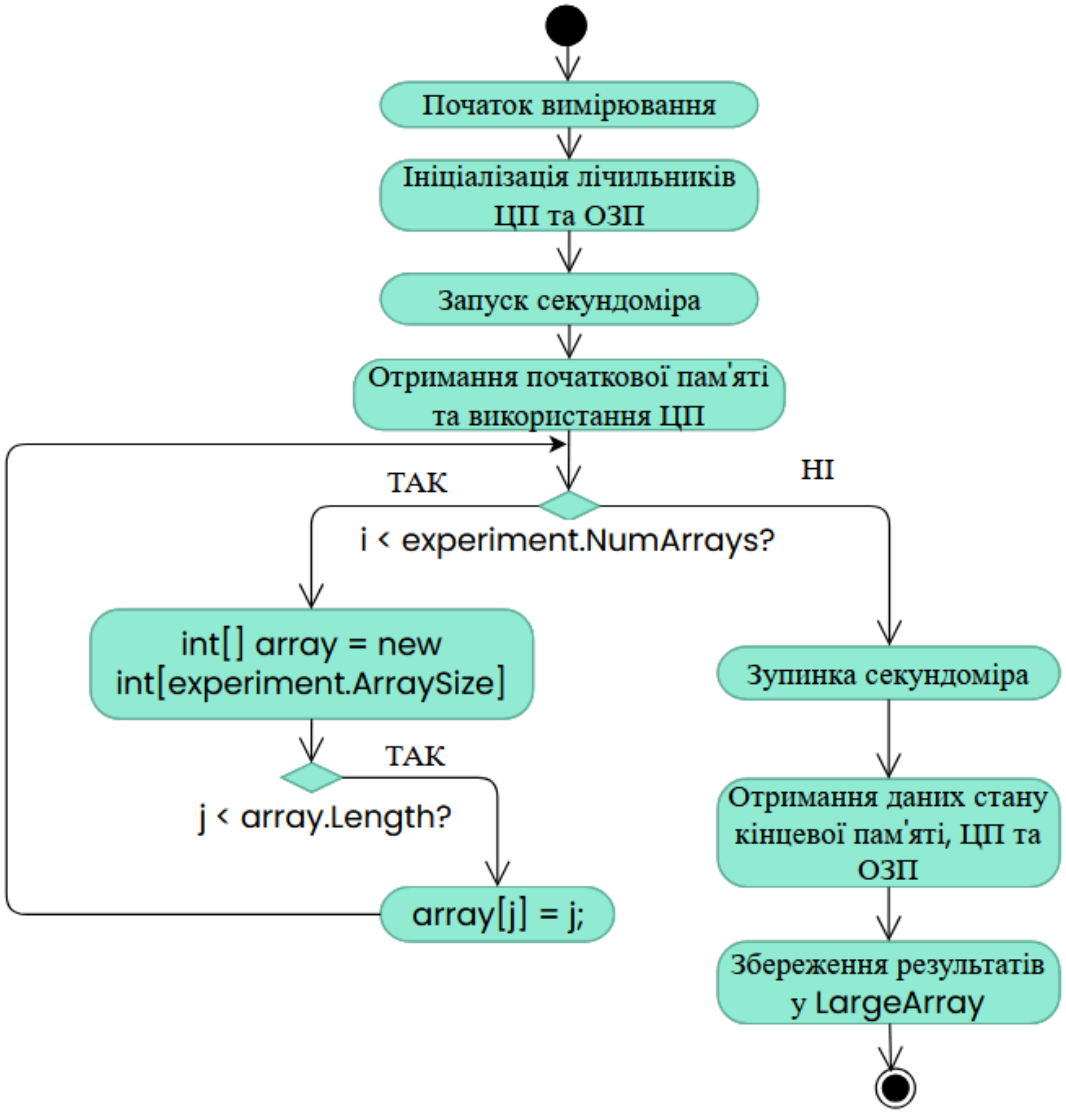

Рисунок 2.4 – Процес вимірювання ефективності для виділення пам’яті та ініціалізації масивів

Процес починається з ініціалізації лічильників центрального процесора та оперативної пам'яті, після чого запускається секундомір для вимірювання часу виконання. Потім відбувається виділення пам'яті під масиви відповідного розміру та їх ініціалізація. Після завершення операцій над масивами секундомір зупиняється, і система отримує кінцеві дані про використання пам'яті та ЦП. Завершальним етапом є збереження результатів у клас LargeArray для подальшого аналізу.

Рис. 2.5 відображає діаграма активності процесу вимірювання параметрів ефективності роботи із небезпечним кодом та покажчиками в ОС Windows.

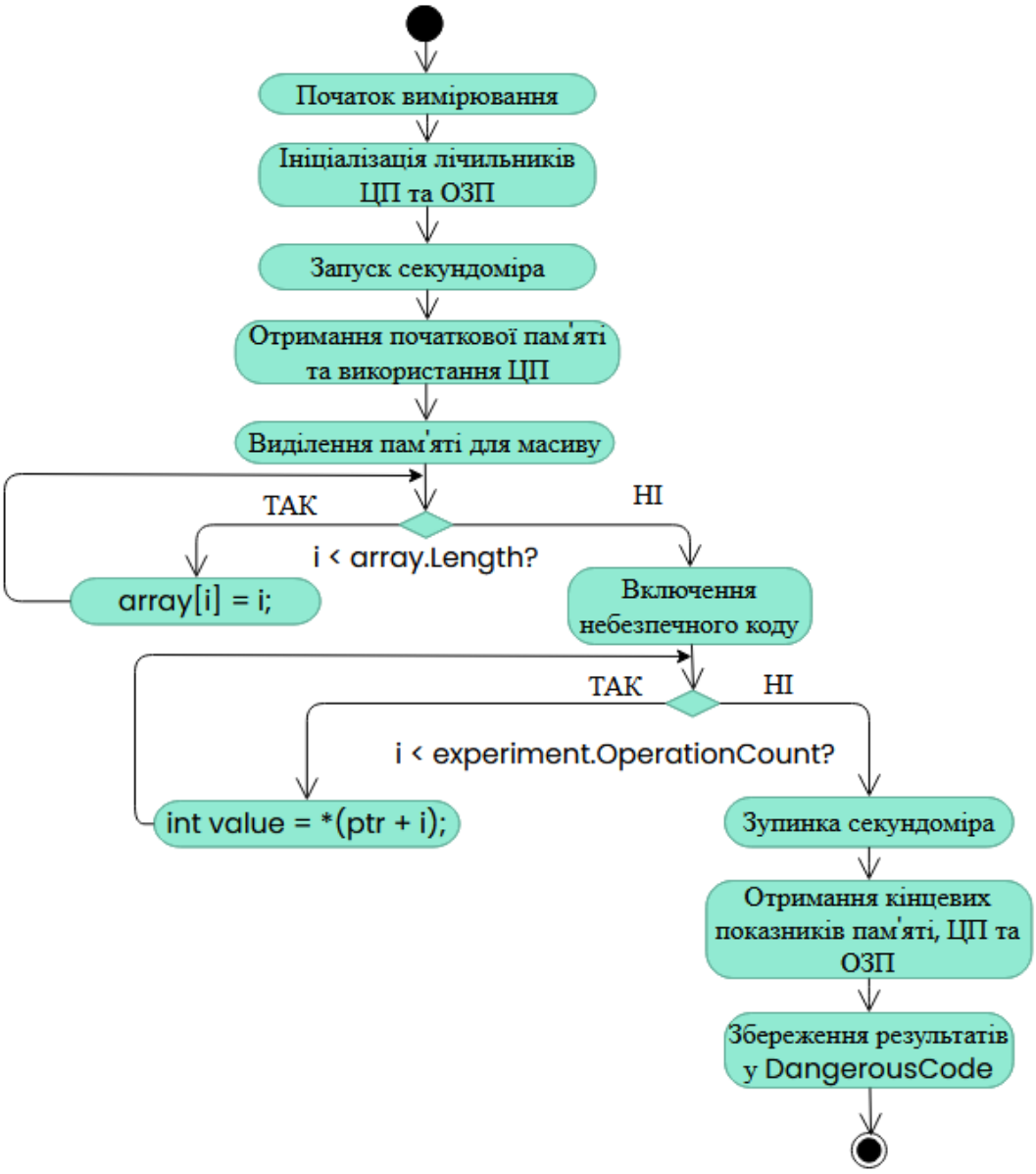

Рисунок 2.5 – Процесу вимірювання параметрів ефективності роботи із небезпечним кодом та покажчиками

На початку система ініціалізує лічильники використання ЦП та ОЗП, а також запускає секундомір для вимірювання часу виконання. Після цього відбувається виділення пам'яті для масиву та його ініціалізація. Якщо масив було повністю проініціалізовано, система переходить до етапу роботи з небезпечним кодом. У цьому випадку використовуються покажчики для доступу до елементів масиву з використанням операцій із покажчиками. Після завершення операцій з небезпечним кодом секундомір зупиняється, і система фіксує кінцеві показники використання

пам'яті, ЦП та ОЗП. Результати експерименту зберігаються в класі DangerousCode для подальшого аналізу.

На рис. 2.6 представлена діаграма активності процесу вимірювання параметрів ефективності роботи з динамічними структурами.

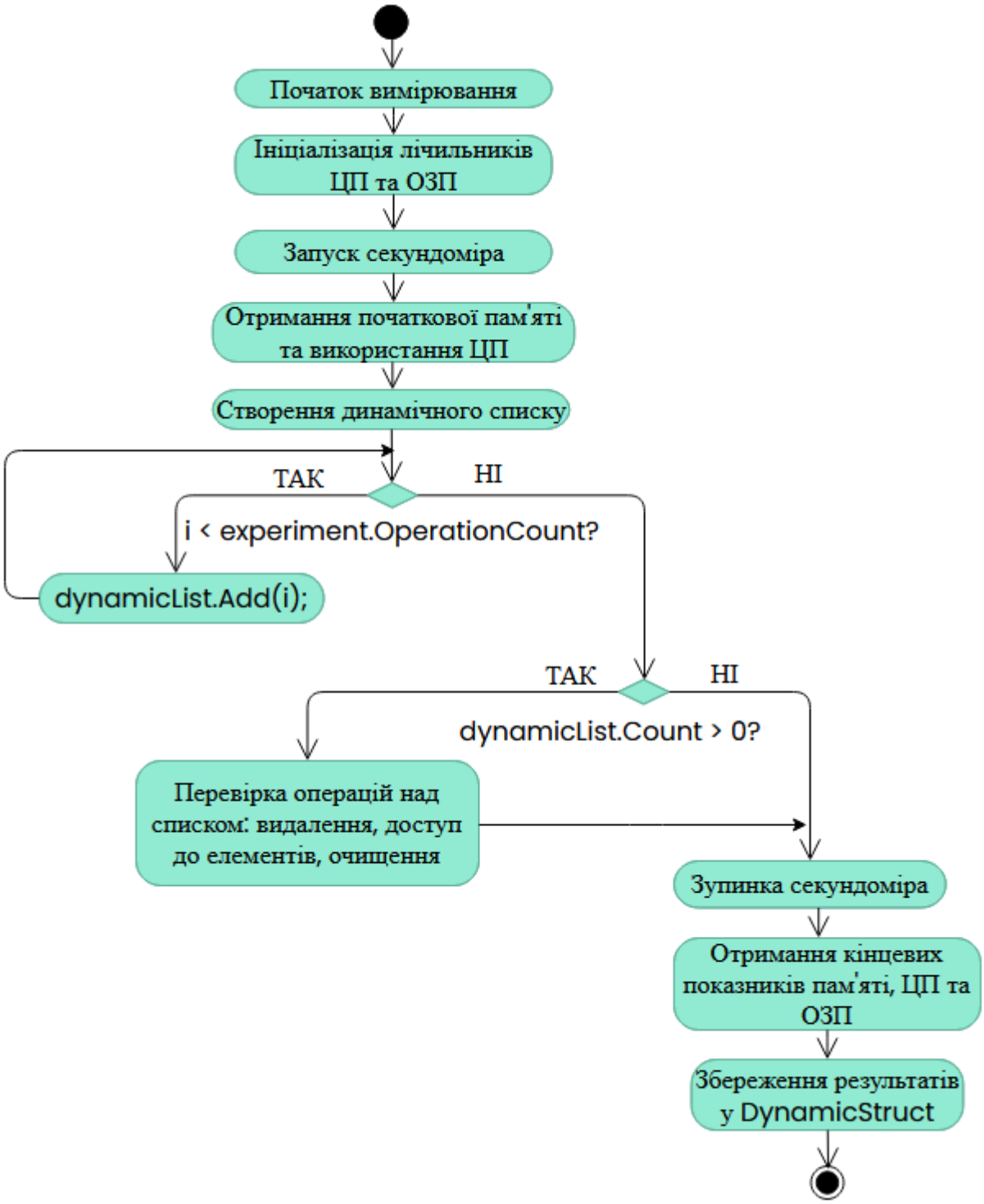

Рисунок 2.6 – Процес вимірювання параметрів ефективності динамічних структур

Система додає елементи до списку, виконує операції над ним, такі як доступ до елементів, їх видалення та очищення. У процесі вимірюється час виконання

операцій, використання пам'яті та навантаження на процесор. Після завершення експерименту результати зберігаються у класі DynamicStruct для подальшого аналізу.

Рис. 2.7 відображає діаграму процесу вимірювання параметрів ефективності роботи із пов'язаним списком.

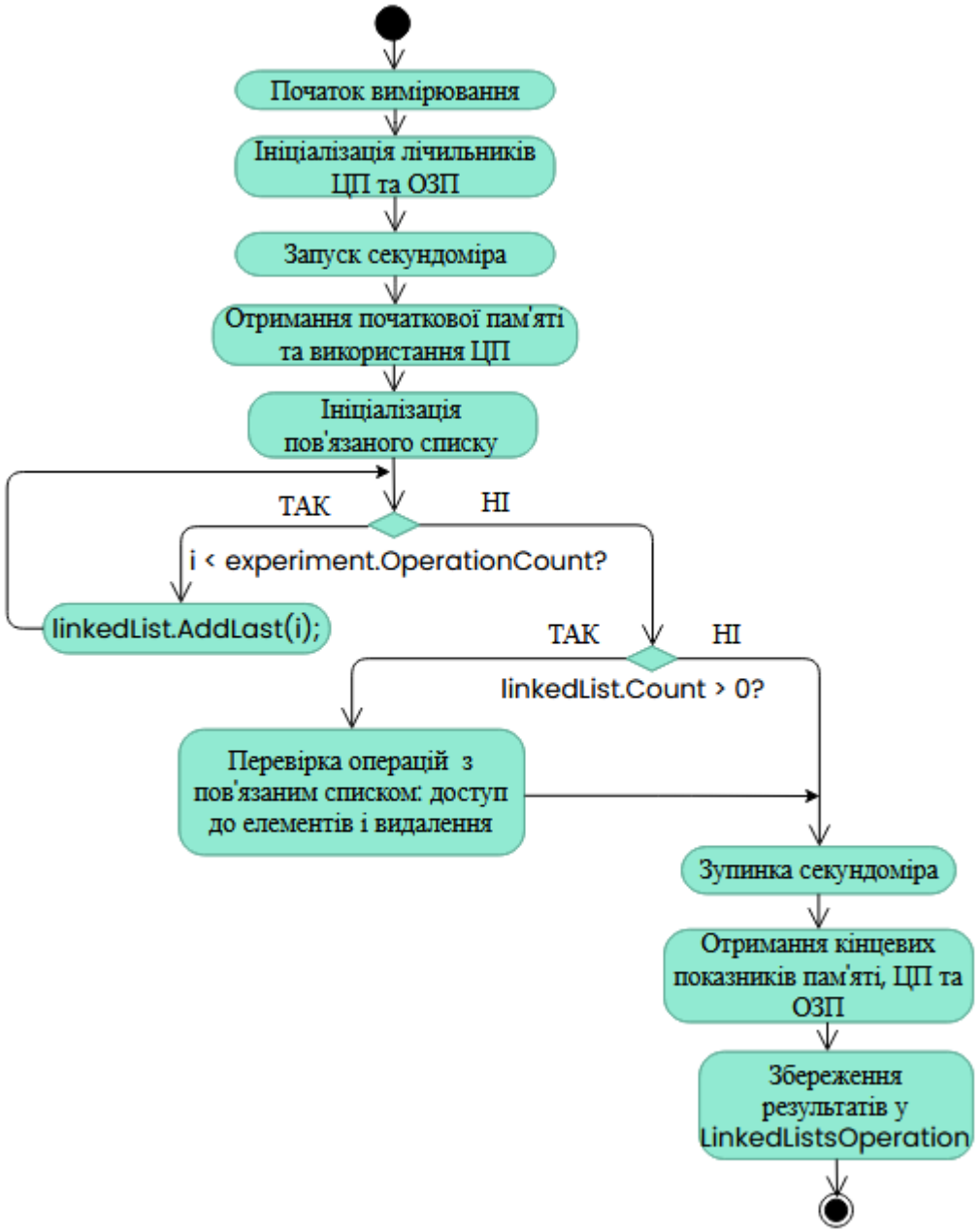

Рисунок 2.7 – Процес вимірювання параметрів ефективності із пов'язаним списком

У процесі вимірювання параметрів ефективності динамічних структур система додає елементи до списку, виконує операції доступу до елементів і їх видалення. У ході процесу відслідковуються показники часу виконання,

використання пам'яті та навантаження на процесор. Після завершення експерименту результати зберігаються у класі LinkedListsOperation для подальшого аналізу.

На рис. 2.8 представлена діаграма активності вимірювання параметрів ефективності роботи із фрагментацією пам’яті в операційній системі.

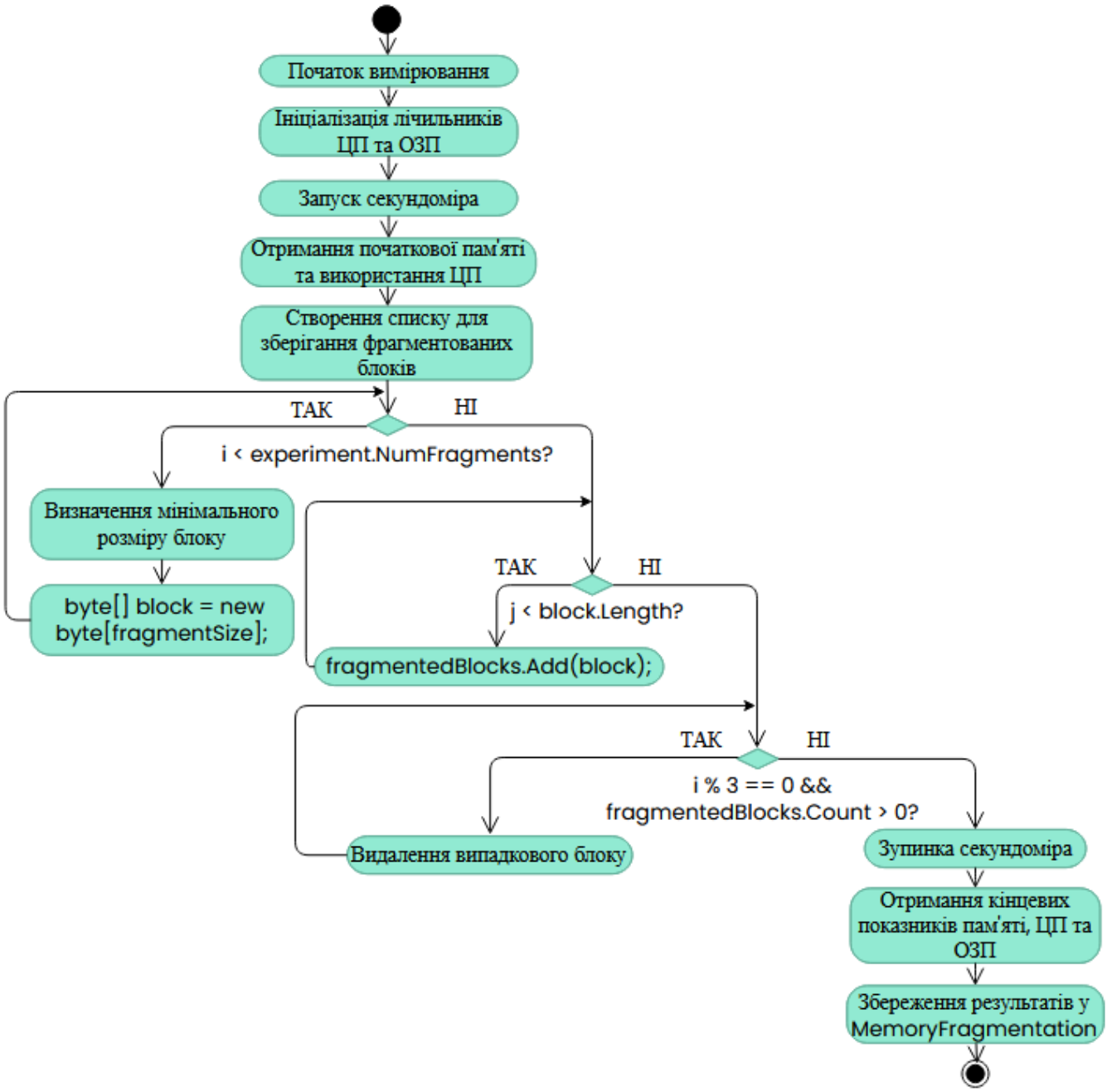

Рисунок 2.8 – Процес вимірювання параметрів ефективності фрагментації пам’яті

В процесі створюються фрагментовані блоки пам'яті, визначаються їх розмір і додає до списку, періодично видаляючи випадкові блоки для симуляції фрагментації. Вимірюються такі параметри, як час виконання операцій,

використання пам'яті та навантаження на процесор. Після завершення експерименту результати зберігаються у класі MemoryFragmentation для подальшого аналізу.

Рис. 2.9 відображає діаграму процесу вимірювання параметрів виділення пам’яті на великій купі об'єктів.

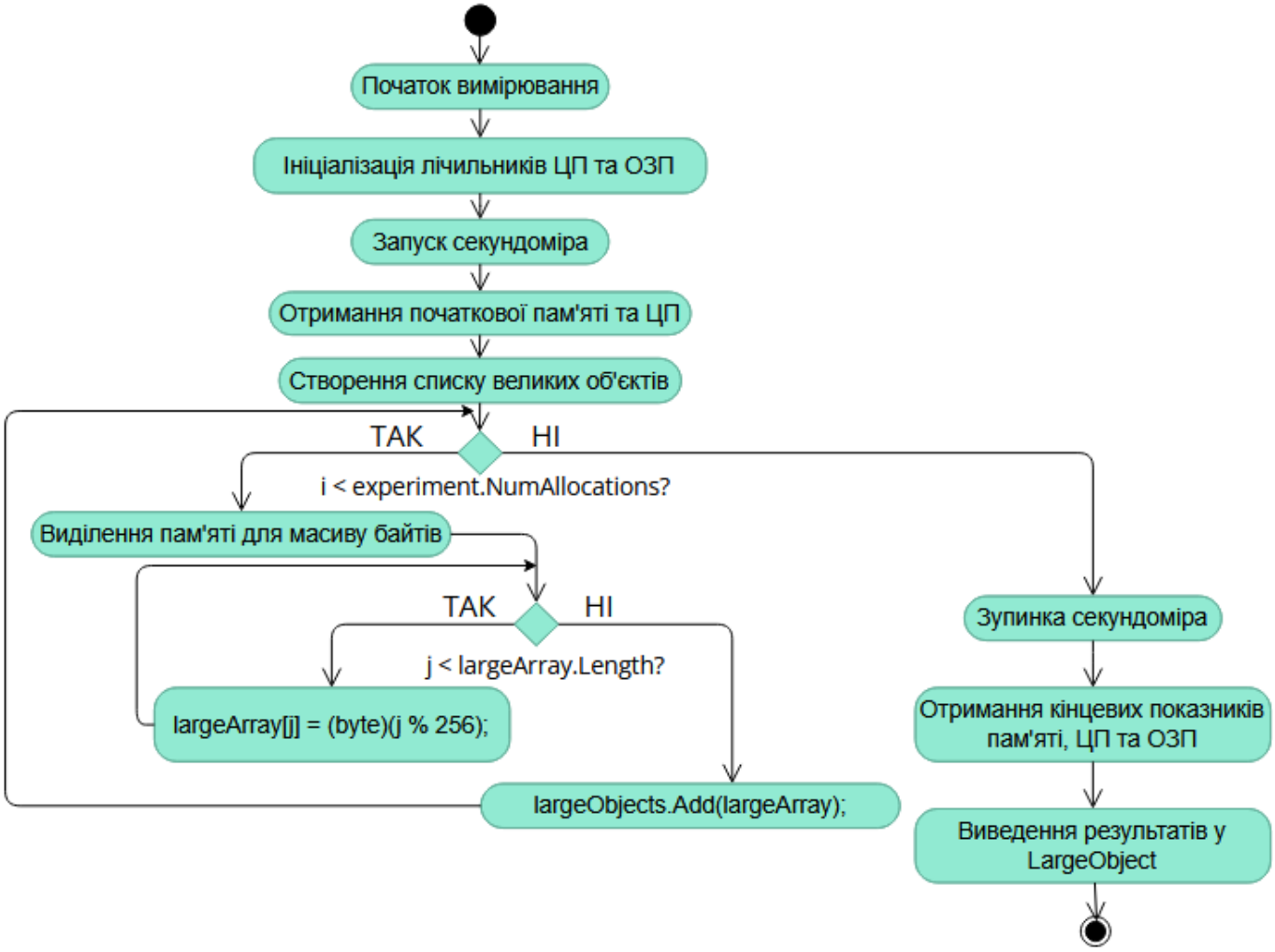

Рисунок 2.9 – Процес вимірювання параметрів ефективності виділення пам’яті на великій купі об'єктів

Ефективність параметрів вимірюється на виділення пам'яті для масивів байтів, в процесі ініціалізується кожен елемент масиву, після чого всі масиви додаються до списку великих об'єктів. Під час цього процесу вимірюються ключові показники, такі як використання пам'яті, час виконання операцій і навантаження на центральний процесор. Після завершення експерименту результати зберігаються у класі LargeObject для подальшого аналізу.

На рис. 2.10 представлена діаграма активності процесу вимірювання параметрів ефективності для операцій пов'язаними з стрічковими маніпуляціями.

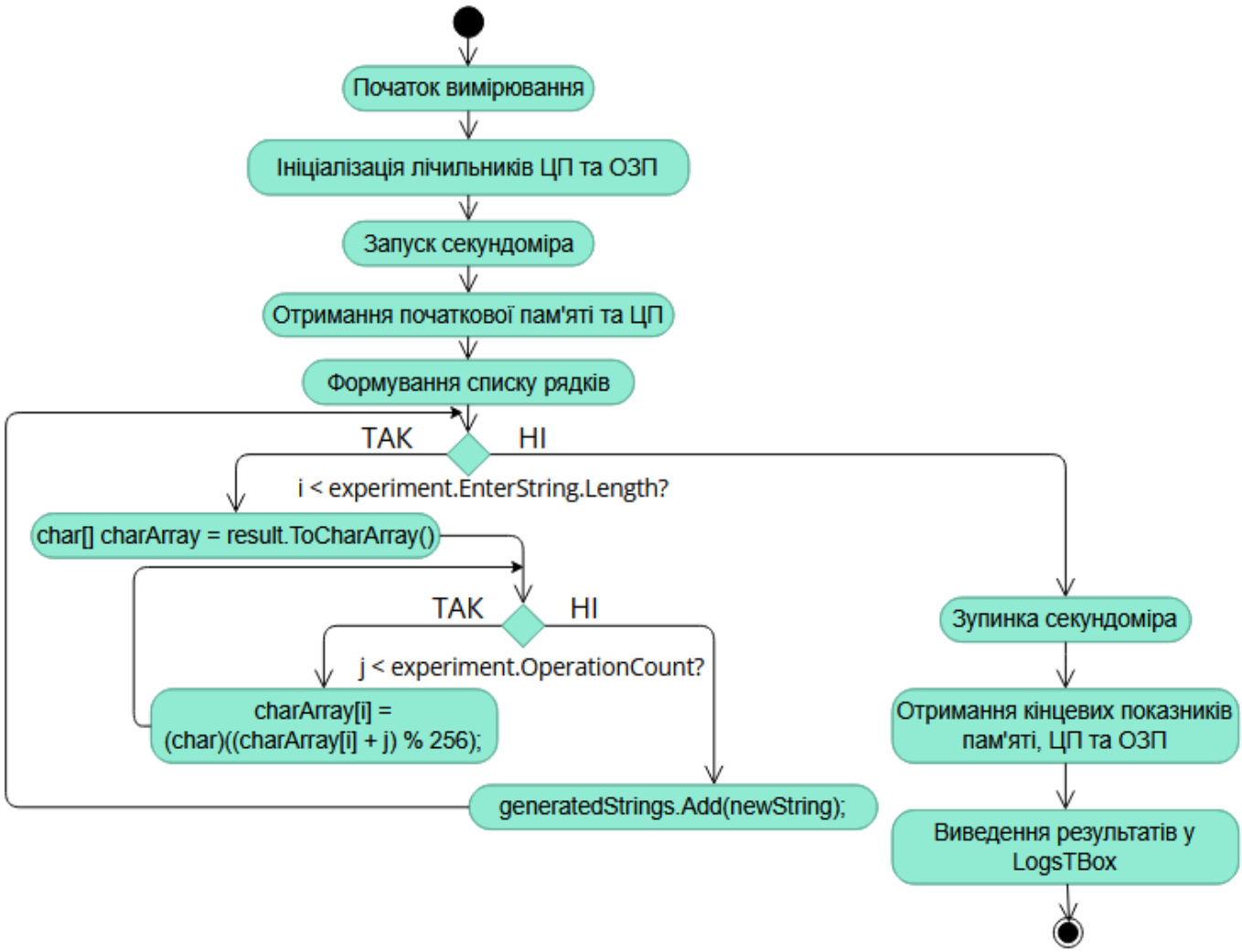

Рисунок 2.10 – Процес вимірювання параметрів ефективності для операцій пов'язаними з стрічковими маніпуляціями

У процесі рядки перетворюються на масив символів, після чого кожен символ змінюється за певним алгоритмом, формуючи нові рядки. Під час виконання операцій вимірюються час виконання, використання пам'яті та навантаження на процесор. Після завершення експерименту результати виводяться до відповідного поля LogsTBox для подальшого аналізу.

Рис. 2.11 відображає діаграму процесу вимірювання параметрів ефективності для операцій пов'язаними із словниками.

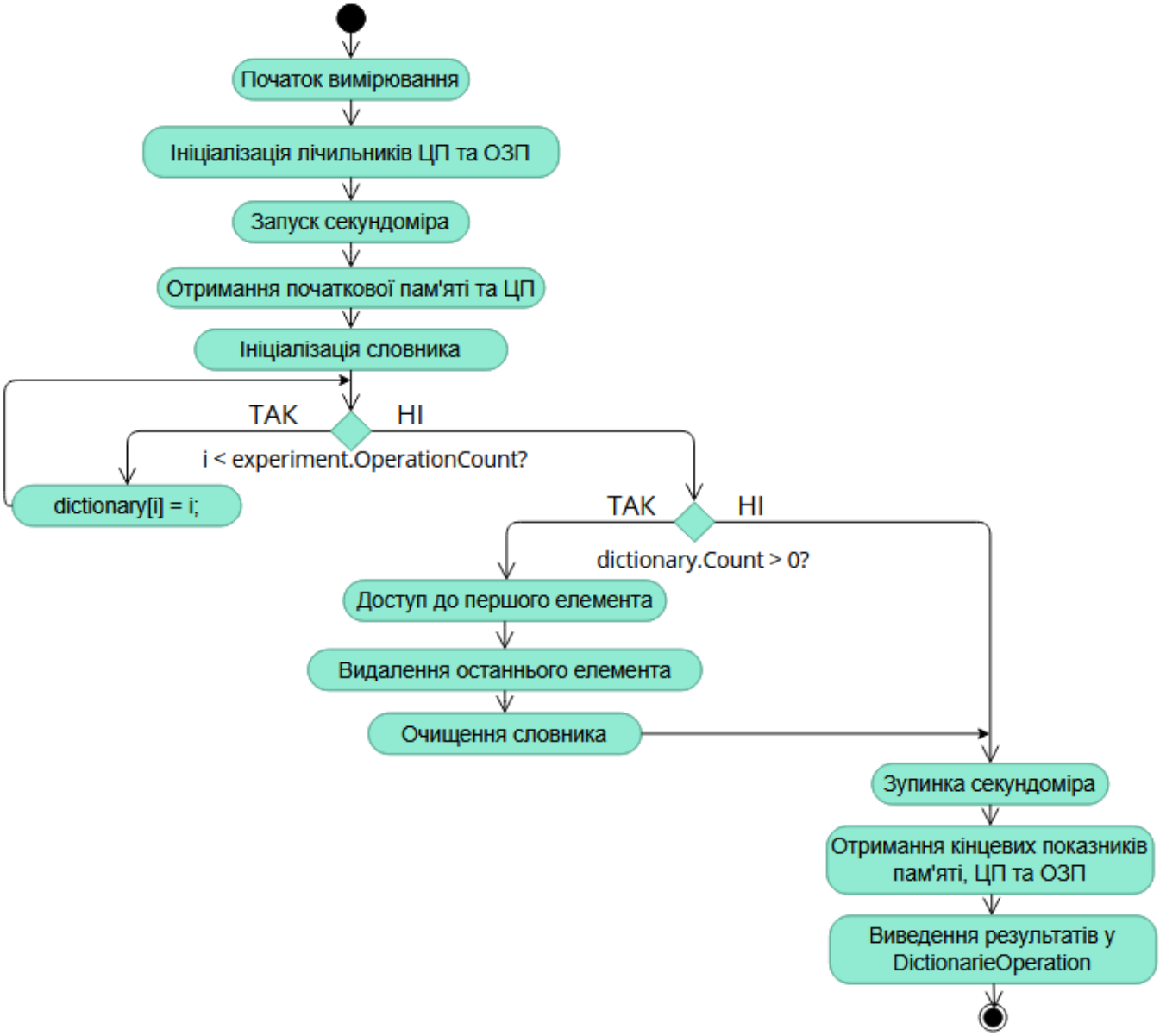

Рисунок 2.11 – Процес вимірювання параметрів ефективності для операцій пов'язаними із словниками

На початку процесу ініціалізується словник і до нього додаються елементи, відповідно до кількості операцій, заданих в експерименті. Після додавання елементів виконується доступ до першого елемента, видалення останнього та очищення всього словника. Під час виконання цих операцій вимірюються час виконання, використання пам'яті та навантаження на процесор. Після завершення експерименту результати зберігаються в класі DictionarieOperation для подальшого аналізу.

Діаграма послідовності показує взаємодію між користувачем і системними компонентами під час виконання певної операції, відображаючи послідовність

повідомлень між об'єктами для досягнення конкретної мети. Вона допомагає візуалізувати, як функціональні блоки системи взаємодіють у часі, дозволяючи чітко зрозуміти, як дані передаються між ними.

На рис. 2.12 зображено діаграму послідовності для роботи користувача із формою для вимірювання параметрів ефективності виділення та ініціалізації масивів в ОС Windows.

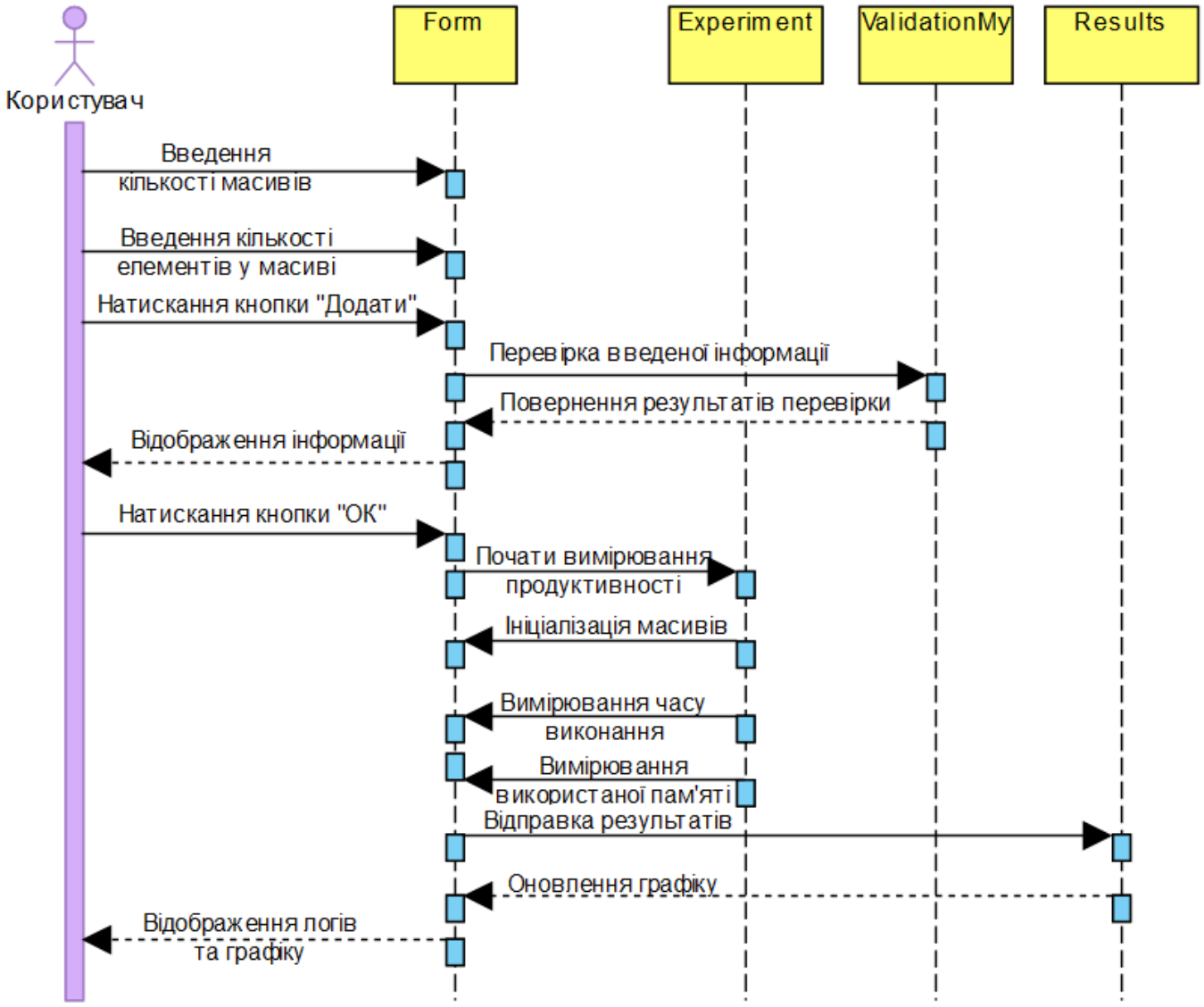

Рисунок 2.12 – Процес роботи користувача із формою «Виділення та ініціалізація масивів»

Користувач вводить кількість масивів та кількість елементів у кожному масиві, після чого натискає кнопку «Додати». Система перевіряє введену інформацію за допомогою класу ValidationMy і повертає результати перевірки. Після цього користувач натискає кнопку «ОК», і система починає вимірювання

продуктивності, ініціалізуючи масиви. Під час виконання тесту система вимірює час виконання, використання пам'яті та відправляє результати для відображення графіків і логів, які показуються користувачу.

Діаграма компонентів використовується для відображення структури системи на рівні взаємодії між різними компонентами. Вона ілюструє, як компоненти пов'язані між собою і як вони взаємодіють під час виконання різних операцій у системі. Ця діаграма допомагає зрозуміти архітектурні зв'язки між різними частинами програми та розподіл відповідальностей між ними.

На рис. 2.13 представлена діаграма компонентів, що демонструє взаємодію між основними складовими системи для дослідження ефективності роботи з пам'яттю в ОС Windows.

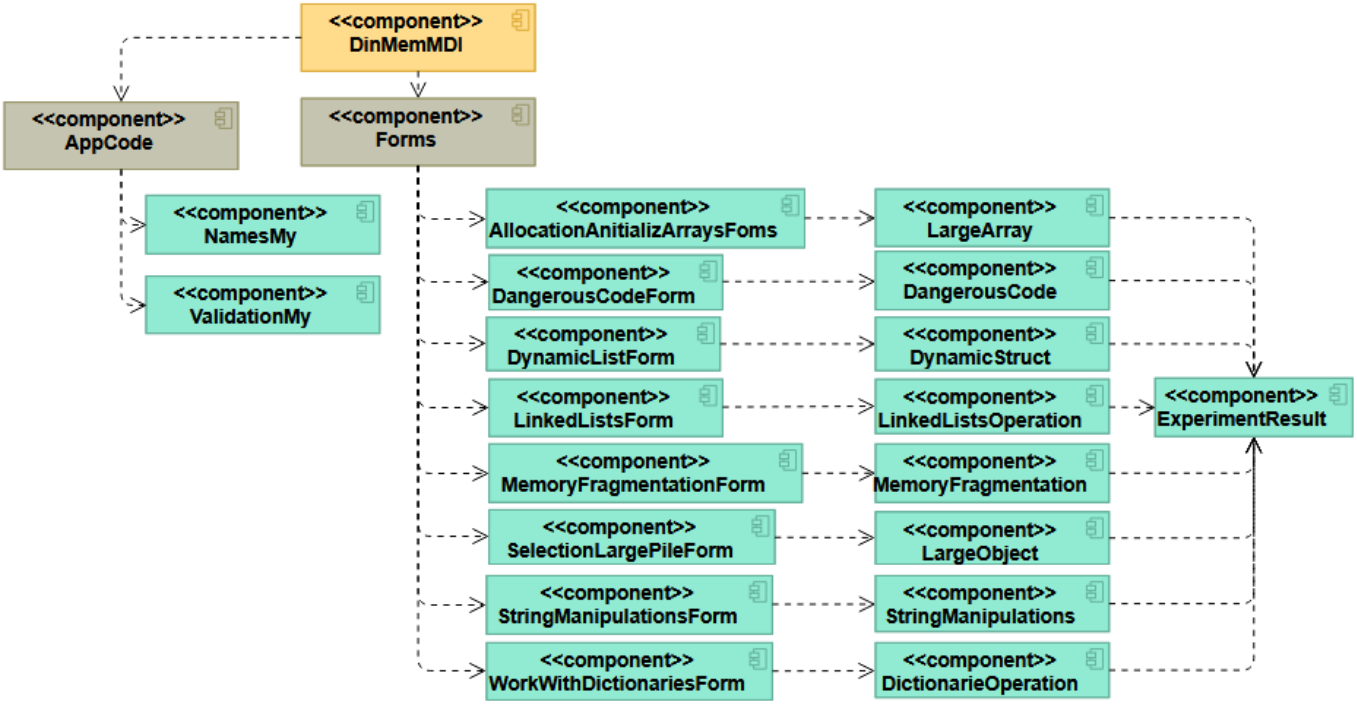

Рисунок 2.13 – Діаграма компонентів системи

Діаграма складається з компонента AppCode, який містить класи NamesMy і ValidationMy, що відповідають за перевірку та обробку даних. Ці компоненти взаємодіють із головним вікном системи – DinMemMDI, яке забезпечує навігацію та управління всіма формами. У системі є кілька форм для різних типів досліджень пам'яті, наприклад, AllocationAnitializArraysForm, DangerousCodeForm, DynamicListForm, та інші. Кожна з цих форм відповідає за певний тип операцій з пам'яттю та пов'язана з відповідними класами, що зберігають результати

експериментів, такими як LargeArray, DangerousCode, DynamicStruct тощо. Всі ці результати об'єднуються через компонент ExperimentResult, що дозволяє збирати й аналізувати дані про продуктивність системи.

2.3 Вибір інструментів для реалізації

При розробці системи для дослідження ефективності роботи з пам'яттю в ОС Windows важливо правильно обрати інструменти, які забезпечать належну продуктивність, надійність і зручність у розробці. Вибір інструментів залежить від кількох факторів: специфікації системи, вимог до продуктивності, підтримки потрібних бібліотек та можливості інтеграції з іншими компонентами. Одним із ключових рішень при розробці є вибір мови програмування, яка найкраще відповідатиме поставленим цілям проєкту.

Python є однією з найпопулярніших мов програмування завдяки простоті синтаксису та великій кількості доступних бібліотек [25]. Ця мова ідеально підходить для швидкого прототипування та створення інструментів для аналізу даних, але може поступатися в продуктивності в порівнянні з іншими мовами. Python часто використовується для розв'язання задач, пов'язаних із науковими дослідженнями, аналізом даних та автоматизацією процесів.

Java – це мова з багатою екосистемою та великою кількістю інструментів для розробки на різних платформах [26]. Вона забезпечує високу продуктивність завдяки технології Java Virtual Machine, що дозволяє ефективно керувати пам'яттю через автоматичний збирач сміття. Java популярна у створенні кросплатформних рішень і корпоративних систем, де важливі надійність і масштабованість.

C# – це потужна мова програмування, що розробляється компанією Microsoft і тісно інтегрована з екосистемою Windows [27]. C# надає ефективні інструменти для розробки як десктопних, так і веб-додатків, забезпечуючи високу продуктивність завдяки своїй оптимізованій роботі з пам'яттю через .NET Framework. Вона є ідеальним вибором для розробки систем, які мають працювати безпосередньо в середовищі Windows.

У табл. 2.3 представлено порівняльний аналіз мов програмування.

Таблиця 2.3 – Порівняльний аналіз мов програмування

Характеристика	Python	Java	C#
Продуктивність	Низька продуктивність через інтерпретовану природу, не підходить для високонавантажених додатків.	Вища продуктивність завдяки JVM, але все ще потребує більше ресурсів для управління пам'яттю.	Висока продуктивність, особливо в середовищі Windows, завдяки інтеграції з .NET і оптимізованим механізмам управління пам'яттю.
Управління пам'яттю	Використовує збирач сміття, але з меншою продуктивністю для операцій з динамічною пам'яттю.	JVM автоматично керує пам'яттю, але має складніший механізм для низькорівневих операцій.	Оптимізоване управління пам'яттю через .NET з можливістю роботи з небезпечним кодом (покажчики) для максимальної продуктивності.
Інтеграція з Windows	Обмежена підтримка, основна екосистема Python не налаштована для Windows-орієнтованих додатків.	Кросплатформна мова, але потребує додаткових налаштувань для оптимальної роботи в середовищі Windows.	Нативна підтримка Windows завдяки тісній інтеграції з операційною системою, що робить C# найкращим вибором для Windows-додатків.
Робота з динамічною пам'яттю	Підтримує динамічну пам'ять, але з меншою оптимізацією та швидкістю.	Добре керує динамічною пам'яттю, але може виникати затримка при великих об'ємах даних через збирач сміття JVM.	Висока ефективність роботи з динамічною пам'яттю, можливість ручного керування і нативної підтримки операцій над пам'яттю через .NET.
Підтримка багатозадачності	Підтримка є, але не так добре підходить для високопродуктивних паралельних операцій.	Висока підтримка багатозадачності через механізми JVM, проте може виникати затримка в складних сценаріях.	Оптимізована для багатозадачності і паралельних обчислень в середовищі Windows, зокрема через Task Parallel Library (TPL) у .NET.

Виходячи з проведеного аналізу, вибір мови програмування C# для розробки системи є оптимальним через її високу продуктивність і тісну інтеграцію з операційною системою Windows. C# пропонує ефективні інструменти для управління динамічною пам'яттю через .NET, що є ключовим для дослідження часової ефективності в ОС Windows. Крім того, можливість роботи з небезпечним кодом і використання покажчиків дозволяє максимально оптимізувати операції з пам'яттю, що є важливим для точного вимірювання продуктивності. Також C# забезпечує високу безпеку та стабільність при виконанні низькорівневих операцій, що є важливим для надійної роботи системи.

Вибір середовища розробки є важливим кроком у процесі створення програмного забезпечення, оскільки воно має забезпечувати зручний інтерфейс, підтримку відповідних мов програмування та інструментів, а також високу продуктивність. Для розробки системи, орієнтованої на дослідження часової ефективності роботи з динамічною пам'яттю в ОС Windows, варто розглянути такі середовища як Visual Studio 2022, Visual Studio Code та Rider. Кожне з них має свої переваги і може бути вибраним залежно від потреб проєкту.

Visual Studio 2022 – це потужне середовище розробки, розроблене Microsoft, що забезпечує повну інтеграцію з .NET та C# [28]. Воно пропонує широкий набір інструментів для розробки як десктопних, так і веб-додатків, надаючи можливості для відлагодження, тестування та оптимізації продуктивності. Visual Studio 2022 підходить для великих проєктів, що вимагають високого рівня інтеграції з Windows та інших технологій Microsoft.

Visual Studio Code – це легке, швидке й розширюване середовище розробки, яке підтримує безліч мов програмування через плагіни [29]. Хоча воно менш потужне, ніж Visual Studio, VS Code є відмінним вибором для проєктів, що потребують швидкої розробки та гнучкої налаштовуваності. Visual Studio Code ідеально підходить для веб-розробки, проте має обмежену підтримку інтегрованих інструментів для розробки великих Windows-додатків.

Rider – це інтегроване середовище розробки від JetBrains, яке підтримує .NET і пропонує чудові інструменти для розробки на C# [30]. Rider поєднує потужний

редактор коду, інструменти для відлагодження та тестування, а також інтелектуальні підказки. Це середовище є альтернативою Visual Studio з акцентом на кросплатформенність, хоча в Windows-середовищі Visual Studio залишається дещо кращим вибором.

У табл. 2.4 представлено порівняння середовищ розробки Visual Studio 2022, Visual Studio Code та Rider за основними характеристиками.

Таблиця 2.4 – Порівняльний аналіз середовищ розробки

Характеристика	Visual Studio 2022	Visual Studio Code	Rider
Інтеграція з Windows та .NET	Найкраща інтеграція з .NET і Windows.	Потребує плагінів для повної підтримки .NET.	Добра підтримка .NET, але менша інтеграція з Windows.
Інструменти для відлагодження	Потужні вбудовані інструменти.	Обмежені можливості, потребує плагінів.	Добрі інструменти, але не настільки глибокі.
Підтримка великих проєктів	Оптимізоване для великих проєктів.	Підходить для невеликих рішень.	Підтримує великі проєкти, але поступається Visual Studio.
Інструменти для тестування	Вбудовані інструменти для тестування.	Потребує плагінів для тестування.	Добрі інструменти, але поступаються Visual Studio.
Візуальний редактор та UI-дизайн	Найкращі інструменти для створення UI.	Обмежені можливості для дизайну UI.	Підтримує UI-дизайн, але поступається Visual Studio.
Підтримка корпоративних рішень	Найкраща підтримка для корпоративних середовищ.	Обмежена підтримка великих рішень.	Підходить для корпоративних рішень, але менш інтегроване з сервісами Microsoft.

Вибір Visual Studio 2022 для розробки системи є оптимальним через її тісну інтеграцію з операційною системою Windows та платформою .NET, що забезпечує ефективну роботу з динамічною пам'яттю та високопродуктивними додатками. Visual Studio 2022 пропонує потужні інструменти для відлагодження, тестування і профілювання, що дозволяє глибоко аналізувати та оптимізувати продуктивність системи.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ЙОГО ТЕСТУВАННЯ

3.1 Реалізація модулів програмного забезпечення

Реалізація модулів програмного забезпечення є ключовим етапом у розробці системи для дослідження часової ефективності роботи з динамічною пам'яттю в ОС Windows. Цей етап охоплює створення окремих функціональних компонентів системи, що відповідають за виконання специфічних завдань: ініціалізацію та виділення пам'яті, роботу з динамічними структурами, обробку даних і вимірювання продуктивності. Кожен модуль виконує певний набір операцій, результати яких дозволяють аналізувати та оцінювати ефективність системи у різних сценаріях.

На основі попередньо розробленої архітектури було реалізовано модулі, що відповідають за окремі операції з пам'яттю, включаючи роботу з масивами, пов'язаними списками, словниками та іншими динамічними структурами. Кожен модуль забезпечує збір даних про час виконання операцій, використання пам'яті та продуктивність процесора, що дозволяє провести детальний аналіз ефективності програмних рішень у різних експериментальних умовах.

Спочатку було реалізовано модуль для вимірювання ефективності виділення пам’яті та ініціалізації масивів у операційній системі Windows, який дозволяє оцінити час і ресурси, необхідні для створення великих масивів. Модуль надає можливість визначати кількість масивів і їхній розмір, після чого здійснюється процес виділення пам'яті та ініціалізація кожного елемента. В ході виконання операцій система збирає інформацію про час виконання, обсяг використаної оперативної пам'яті та навантаження на процесор.

Окрім того, модуль реалізований таким чином, що він дозволяє аналізувати різні параметри, що впливають на ефективність виділення пам'яті, зокрема різні розміри масивів і кількість одночасно ініціалізованих структур. Результати вимірювань фіксуються і відображаються у вигляді графіків, що допомагає краще оцінити загальну продуктивність системи під час роботи з динамічною пам'яттю. Це забезпечує більш глибоке розуміння того, як ОС Windows обробляє великі об'єми даних і які ресурси використовуються при цьому.

Код, приведений на рис. 3.1 відповідає за обробку події при натисканні кнопки додавання нового елемента типу LargeArray. Спочатку перевіряється коректність введених даних за допомогою методу «IsDataEnteringCorrecting». Якщо дані вірні, створюється новий об'єкт типу LargeArray, після чого встановлюються його властивості на основі даних, введених у текстових полях. Зокрема, кількість масивів і розмір масиву отримуються з відповідних текстових полів і конвертуються у числовий формат. Далі цей новий об'єкт додається до списку _AllLargeArray, і оновлюється відображення цього списку в таблиці. Після успішного додавання даних текстові поля очищуються за допомогою методу «CliarData».

```
private void AddNewElementBtn_Click(object sender, EventArgs e) {
  if (IsDataEnteringCorrecting()) {
    LargeArray oneLargeArray = new LargeArray();
    oneLargeArray.NumArrays = Convert.ToInt32(NumArraysTBox.Text);
    oneLargeArray.ArraySize = Convert.ToInt32(ArraySizeTBox.Text);
    _AllLargeArray.Add(oneLargeArray);
    LoadDataInLargeArrayGW(_AllLargeArray);
    CliarData();
  }
}
```

Рисунок 3.1 – Додавання нового елемента типу LargeArray

Наступним кроком реалізовано код, який обробляє подію кліку на клітинку в таблиці LargeArrayGridView та видаляє обраний елемент із списку (рис. 3.2).

```
private void LargeArrayGridView_CellClick(object sender, DataGridViewCellEventArgs e) {
  if (e.ColumnIndex == 3 && LargeArrayGridView[0, e.RowIndex].Value.ToString()
    != _AllLargeArray[0].Message) {
    int num = Convert.ToInt32(LargeArrayGridView[0, e.RowIndex].Value.ToString());
    for (int i = 0; i < _AllLargeArray.Count; i++) {
      if (num == _AllLargeArray[i].Number) {
        _AllLargeArray.RemoveAt(i);
        break;
      }
    }
    LoadDataInLargeArrayGW(_AllLargeArray);
  }
}
```

Рисунок 3.2 – Видалення елемента типу LargeArray

Якщо користувач натискає на клітинку в колонці з індексом 3, і значення в першій колонці рядка не збігається зі значенням повідомлення першого елемента списку _AllLargeArray, система виконує подальші дії. Спочатку отримується номер масиву з першої клітинки цього рядка і конвертується в числовий формат. Далі виконується пошук цього номера в списку _AllLargeArray, після чого відповідний

елемент видаляється зі списку. Після видалення елемента таблиця оновлюється за допомогою методу «LoadDataInLargeArrayGW».

```
private void TestBtn_Click(object sender, EventArgs e) {
  if (_AllLargeArray.Count >= 2) {
    // Очищаємо логи перед початком нових експериментів
    LogsTBox.Clear();
    // Ініціалізуємо графік перед проведенням експериментів
    InitializeChart();
    // Проведення серії експериментів
    PerformExperiments(_AllLargeArray);
    // Оновлюємо дані графіка після завершення експериментів
    UpdateChartWithData();
  } else {
    MessageBox.Show(NamesMy.Messages.InsufficientData, "Увага");
  }
}
```

Рисунок 3.3 – Додавання нового елемента типу LargeArray

Для запуску процедури вимірювання ефективності виділення пам’яті та ініціалізації масивів реалізовано код, який викликається при натисканні на кнопку «Test» (рис. 3.4).

```
private void TestBtn_Click(object sender, EventArgs e) {
  if (_AllLargeArray.Count >= 2) {
    // Очищаємо логи перед початком нових експериментів
    LogsTBox.Clear();
    // Ініціалізуємо графік перед проведенням експериментів
    InitializeChart();
    // Проведення серії експериментів
    PerformExperiments(_AllLargeArray);
    // Оновлюємо дані графіка після завершення експериментів
    UpdateChartWithData();
  } else {
    MessageBox.Show(NamesMy.Messages.InsufficientData, "Увага");
  }
}
```

Рисунок 3.4 – Запуск процедури вимірювання ефективності виділення пам’яті та ініціалізації масивів

На початку перевіряється, чи містить список _AllLargeArray принаймні два елементи. Якщо умова виконується, очищується поле для логів LogsTBox, щоб звільнити його перед новими експериментами. Далі ініціалізується графік за допомогою методу «InitializeChart», готуючи його для відображення результатів. Після цього запускається основний процес проведення експериментів, де результати зберігаються і аналізуються. Після завершення експериментів графік оновлюється відповідно до отриманих даних. Якщо ж у списку недостатньо даних для

проведення тестування, виводиться повідомлення про помилку з інформативним текстом «Увага».

Для початкової підготовки графіків для відображення результатів експериментів реалізовано метод «InitializeChart», код якого представлено на рис. 3.5.

```
private void InitializeChart() {
    // Очищуємо серії та осі
    GraphicsCC.Series.Clear();
    GraphicsCC.AxisX.Clear();
    GraphicsCC.AxisY.Clear();
    // Створюємо серії для графіка
    var executionTimeSeries = new LineSeries {
        Title = "Час виконання",
        Values = new ChartValues<double>()
    };
    var memoryUsedSeries = new LineSeries {
        Title = "Використана пам'ять (MB)",
        Values = new ChartValues<double>()
    };
    // Додаємо серії на графік
    GraphicsCC.Series.Add(executionTimeSeries);
    GraphicsCC.Series.Add(memoryUsedSeries);
    // Додаємо підписи для осі X
    GraphicsCC.AxisX.Add(new LiveCharts.Wpf.Axis {
        Title = "Кількість масивів",
        Labels = _AllLargeArray.Select(ex => ex.NumArrays.ToString()).ToArray(),
        Separator = new LiveCharts.Wpf.Separator { Step = 1 }
    });
    // Вісь Y
    GraphicsCC.AxisY.Add(new LiveCharts.Wpf.Axis {
        Title = "Час виконання (мс)"
    });
}
```

Рисунок 3.5 – Код методу для ініціалізації графіків

На старті методу очищуються всі серії даних і осі, щоб забезпечити, що попередні дані не заважають новим результатам. Потім створюються дві нові серії для графіка: одна для часу виконання операцій і друга для використаної пам'яті. Ці серії додаються до графіка, щоб відобразити результати майбутніх експериментів. Для осі X задаються підписи з кількістю масивів, використаних в експериментах, з чіткими відмітками для кожного значення. Вісь Y відповідає за відображення часу виконання операцій у мілісекундах, що надає користувачу можливість проаналізувати ефективність проведених операцій на графіку.

Метод, представлений на рис. 3.6 використовується для оновлення графіка з новими даними після проведення експериментів. Спочатку перевіряється, чи на графіку вже існують серії даних для оновлення. Якщо серії присутні, отримуються

об'єкти для серій, що відповідають за час виконання операцій та використану пам'ять. Старі дані, які могли залишитися після попередніх експериментів, очищуються, щоб уникнути змішування результатів. Далі, для кожного експерименту зі списку _AllLargeArray, у відповідні серії додаються нові значення: час виконання операцій у мілісекундах та обсяг використаної пам'яті в мегабайтах. Це забезпечує актуалізацію графіка, дозволяючи користувачу переглянути результати експериментів у візуальному форматі.

```
private void UpdateChartWithData() {
    // Переконуємося, що є серії для оновлення
    if (GraphicsCC.Series.Count > 0) {
        var executionTimeSeries = (LineSeries)GraphicsCC.Series[0];
        var memoryUsedSeries = (LineSeries)GraphicsCC.Series[1];
        // Очищуємо старі дані
        executionTimeSeries.Values.Clear();
        memoryUsedSeries.Values.Clear();
        // Додаємо нові дані
        foreach (var experiment in _AllLargeArray) {
            executionTimeSeries.Values.Add(experiment.ExecutionTimeMilliseconds);
            memoryUsedSeries.Values.Add(experiment.MemoryUsedMB);
        }
    }
}
```

Рисунок 3.6 – Код методу для оновлення графіка

Для вимірювання операцій виділення та ініціалізації масивів і збереження результатів у об'єкті типу LargeArray, реалізовано метод «MeasureLargeArrayAllocation».

```
private void MeasureLargeArrayAllocation(LargeArray experiment) {
    PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time", "_Total");
    PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    long initialMemory = GC.GetTotalMemory(true);
    double initialCpuUsage = cpuCounter.NextValue();
    for (int i = 0; i < experiment.NumArrays; i++) {
        int[] array = new int[experiment.ArraySize];
        for (int j = 0; j < array.Length; j++) {
            array[j] = j; // Ініціалізація масиву
        }
    }
    stopwatch.Stop();
    long finalMemory = GC.GetTotalMemory(false);
    double finalCpuUsage = cpuCounter.NextValue();
    double availableMemoryMB = ramCounter.NextValue();
    experiment.ExecutionTimeMilliseconds = stopwatch.Elapsed.TotalMilliseconds;
    experiment.MemoryUsedMB = (finalMemory - initialMemory) / (1024 * 1024);
    experiment.AverageCpuUsagePercent = (initialCpuUsage + finalCpuUsage) / 2;
    experiment.AvailableMemoryMB = availableMemoryMB;
}
```

Рисунок 3.7 – Код методу «MeasureLargeArrayAllocation»

На початку методу створюються лічильники для вимірювання завантаження процесора та обсягу доступної пам'яті. За допомогою секундоміра вимірюється час виконання операцій. Після старту секундоміра зчитується початковий обсяг пам'яті та завантаження процесора, а потім у циклі створюються масиви з певною кількістю елементів, які заповнюються індексами. Після завершення роботи секундомір зупиняється, знову фіксуються показники пам'яті та процесора. Результати вимірювань, включаючи час виконання, використану пам'ять та середнє навантаження на процесор, зберігаються в об'єкті експерименту для подальшого аналізу.

Метод «DisplayResults», код якого представлений на рис. 3.8 використовується для відображення результатів експерименту в текстовому полі LogsTBox. Замість виведення інформації у консоль, результати додаються безпосередньо в інтерфейс користувача.

```
private void DisplayResults(ExperimentResult result) {
  // Виведення результатів у LogsTBox замість Console
  LogsTBox.AppendText($"Час виконання: " +
    $"{result.ExecutionTimeMilliseconds} мс{Environment.NewLine}");
  LogsTBox.AppendText($"Використано пам'яті:" +
    $" {result.MemoryUsedMB} MB{Environment.NewLine}");
  LogsTBox.AppendText($"Середнє навантаження процесора: " +
    $"{result.AverageCpuUsagePercent}%{Environment.NewLine}");
  LogsTBox.AppendText($"Доступна пам'ять: " +
    $"{result.AvailableMemoryMB} MB{Environment.NewLine}");
  Application.DoEvents();
}
```

Рисунок 3.8 – Код методу «DisplayResults»

Метод отримує об'єкт типу ExperimentResult, після чого виводить ключові показники: час виконання операцій, обсяг використаної пам'яті, середнє навантаження на процесор та кількість доступної пам'яті. Кожен результат додається у вигляді окремого рядка в текстове поле LogsTBox, після чого викликається метод «Application.DoEvents» для оновлення інтерфейсу в реальному часі.

На рис. 3.9 представлено метод «PerformExperiments», який відповідає за виконання серії експериментів із динамічним оновленням прогрес-бару для відображення ходу виконання. Спочатку встановлюється максимальне значення прогрес-бару відповідно до кількості експериментів, а його значення скидається на

початкове. Далі, для кожного експерименту відображається номер і коротке повідомлення в текстовому полі LogsTBox перед запуском. Викликається метод для проведення вимірювань виділення пам'яті для кожного експерименту, після чого результати відображаються в інтерфейсі. По завершенню кожного експерименту прогрес-бар оновлюється, а інтерфейс перерисовується для реального відображення процесу за допомогою «Application.DoEvents».

```
private void PerformExperiments(List<LargeArray> experiments) {
  TestPBar.Maximum = experiments.Count; // Встановлюємо максимальне значення для прогрес-бару
  TestPBar.Value = 0; // Скидаємо прогрес на початок
  foreach (var experiment in experiments) {
    // Виведення номера експерименту та повідомлення перед запуском
    LogsTBox.AppendText($"Експеримент {experiment.Number}:" +
      $" {experiment.Message}{Environment.NewLine}");
    // Проведення заміру і збереження результатів
    MeasureLargeArrayAllocation(experiment);
    // Виведення результатів після завершення кожного експерименту
    DisplayResults(experiment);
    // Оновлюємо прогрес бар
    TestPBar.Value += 1;
    // Оновлюємо інтерфейс, щоб прогрес бар відображався під час виконання експериментів
    Application.DoEvents();
  }
}
```

Рисунок 3.9 – Код методу «PerformExperiments»

Для перевірки введення даних користувачем у текстових полях для кількості масивів та їх розміру було реалізовано метод «IsDataEnteringCorrecting» (рис. 3.10).

```
private bool IsDataEnteringCorrecting() {
  bool isCorrect = true;
  if (_Validation.IsDataConvertToInt(NumArraysTBox.Text)) {
    if (_Validation.IsDataInThisScope(1, 10000000, Convert.ToInt32(NumArraysTBox.Text))) {
      NumArraysValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
    } else {
      NumArraysValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
      isCorrect = false;
    }
  } else {
    NumArraysValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
    isCorrect = false;
  }
  if (_Validation.IsDataConvertToInt(ArraySizeTBox.Text)) {
    if (_Validation.IsDataInThisScope(1, 10000000, Convert.ToInt32(ArraySizeTBox.Text))) {
      ArraySizeValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
    } else {
      ArraySizeValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
      isCorrect = false;
    }
  } else {
    ArraySizeValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
    isCorrect = false;
  }
  return isCorrect;
}
```

Рисунок 3.10 – Код методу перевірки введених даних

Метод спочатку встановлює початкове значення, яке вказує на коректність введених даних. Для перевірки поля кількості масивів спочатку викликається метод, який перевіряє, чи можна перетворити введене значення на ціле число. Якщо дані можуть бути перетворені, наступним кроком є перевірка, чи знаходяться вони в межах від 1 до 10 мільйонів. У випадку, якщо всі перевірки пройдені, виводиться повідомлення про успішність валідації. Якщо ж виникає помилка на будь-якому з етапів, відповідні повідомлення про помилку виводяться на мітки, а прапорець коректності встановлюється на хибний. Аналогічні перевірки виконуються для поля розміру масивів. Наприкінці метод повертає результат перевірки, вказуючи, чи є введені дані коректними для подальшого виконання програми.

Для вимірювання ефективності роботи з небезпечним кодом і покажчиками під час виконання операцій з масивом реалізовано метод «MeasureDangerousCodeAllocation» (рис. 3.11).

```
private unsafe void MeasureDangerousCodeAllocation(DangerousCode experiment) {
    PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time", "_Total");
    PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
    Stopwatch stopwatch = new Stopwatch();
    long initialMemory = GC.GetTotalMemory(true);
    double initialCpuUsage = cpuCounter.NextValue();
    // Виділення пам'яті на масив
    int[] array = new int[experiment.OperationCount];
    // Ініціалізація масиву
    for (int i = 0; i < array.Length; i++) {
        array[i] = i;
    }
    // Доступ за допомогою небезпечного коду (покажчики)
    unsafe {
        fixed (int* ptr = array) {
            stopwatch.Start();
            for (int i = 0; i < experiment.OperationCount; i++) {
                // Доступ до елементів через покажчик
                int value = *(ptr + i);
            }
            stopwatch.Stop();
        }
    }
    long finalMemory = GC.GetTotalMemory(false);
    double finalCpuUsage = cpuCounter.NextValue();
    double availableMemoryMB = ramCounter.NextValue();
    // Збереження результатів
    experiment.ExecutionTimeMilliseconds = stopwatch.Elapsed.TotalMilliseconds;
    experiment.MemoryUsedMB = (finalMemory - initialMemory) / (1024 * 1024);
    experiment.AverageCpuUsagePercent = (initialCpuUsage + finalCpuUsage) / 2;
    experiment.AvailableMemoryMB = availableMemoryMB;
}
```

Рисунок 3.11 – Код методу «MeasureDangerousCodeAllocation»

Спочатку створюються лічильники для відстеження завантаження процесора та доступної пам'яті, а також ініціалізується секундомір для вимірювання часу виконання. Після цього відбувається виділення пам'яті для масиву і його ініціалізація значеннями, де кожен елемент масиву отримує своє значення за індексом. Для доступу до елементів масиву використовується небезпечний код – через покажчики. У блоці unsafe за допомогою оператора fixed масив фіксується в пам'яті, і відбувається послідовний доступ до елементів через покажчик. Секундомір фіксує час, необхідний для доступу до елементів масиву. Після завершення операцій знову отримуються показники пам'яті та завантаження процесора. Результати вимірювань, включаючи час виконання, використану пам'ять та середнє навантаження на процесор, зберігаються в об'єкті експерименту для подальшого аналізу.

Метод на рис. 3.12 виконує вимірювання ефективності операцій з динамічними структурами, використовуючи для цього динамічний список.

```
private void MeasureDynamicStructAllocation(DynamicStruct experiment) {
    PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time", "_Total");
    PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    long initialMemory = GC.GetTotalMemory(true);
    double initialCpuUsage = cpuCounter.NextValue();
    // Операції з динамічним списком
    List<int> dynamicList = new List<int>();
    for (int i = 0; i < experiment.OperationCount; i++) {
        // Додаємо елементи до динамічного списку
        dynamicList.Add(i);
    }
    // Перевірка операцій над списком: видалення, доступ до елементів, очищення
    if (dynamicList.Count > 0) {
        // Доступ до першого елемента
        int firstElement = dynamicList[0];
        // Видалення останнього елемента
        dynamicList.RemoveAt(dynamicList.Count - 1);
        // Очищення списку
        dynamicList.Clear();
    }
    stopwatch.Stop();
    long finalMemory = GC.GetTotalMemory(false);
    double finalCpuUsage = cpuCounter.NextValue();
    double availableMemoryMB = ramCounter.NextValue();
    experiment.ExecutionTimeMilliseconds = stopwatch.Elapsed.TotalMilliseconds;
    experiment.MemoryUsedMB = (finalMemory - initialMemory) / (1024 * 1024);
    experiment.AverageCpuUsagePercent = (initialCpuUsage + finalCpuUsage) / 2;
    experiment.AvailableMemoryMB = availableMemoryMB;
}
```

Рисунок 3.12 – Код методу «MeasureDynamicStructAllocation»

Лічильники збирають дані про завантаження процесора та доступну пам'ять, а секундомір фіксує час виконання. Після цього створюється динамічний список і до нього додаються елементи на основі кількості операцій, визначеної в експерименті. Далі виконуються перевірки на роботу зі списком: доступ до першого елемента, видалення останнього і повне очищення списку. Після зупинки секундоміра отримуються кінцеві показники завантаження процесора та об'єму використаної пам'яті. Дані про час виконання, обсяг пам'яті та середнє навантаження на процесор зберігаються в об'єкті експерименту для подальшого аналізу продуктивності.

На рис. 3.13 представлено метод, який відповідає за вимірювання ефективності операцій з пов'язаним списком. Лічильники збирають дані про завантаження процесора і доступну пам'ять, а секундомір відслідковує тривалість операцій.

```
private void MeasureLinkedListOperations(LinkedListsOperation experiment) {
    PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time", "_Total");
    PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    long initialMemory = GC.GetTotalMemory(true);
    double initialCpuUsage = cpuCounter.NextValue();
    // Ініціалізація пов'язаного списку
    LinkedList<int> linkedList = new LinkedList<int>();
    // Додавання елементів у пов'язаний список
    for (int i = 0; i < experiment.OperationCount; i++) {
        linkedList.AddLast(i); // Додаємо елемент в кінець списку
    }
    // Операції з пов'язаним списком: доступ до елементів і видалення
    if (linkedList.Count > 0) {
        // Доступ до першого елемента
        int firstElement = linkedList.First.Value;
        // Видалення останнього елемента
        linkedList.RemoveLast();
        // Очищення списку
        linkedList.Clear();
    }
    stopwatch.Stop();
    long finalMemory = GC.GetTotalMemory(false);
    double finalCpuUsage = cpuCounter.NextValue();
    double availableMemoryMB = ramCounter.NextValue();
    // Збереження результатів експерименту
    experiment.ExecutionTimeMilliseconds = stopwatch.Elapsed.TotalMilliseconds;
    experiment.MemoryUsedMB = (finalMemory - initialMemory) / (1024 * 1024);
    experiment.AverageCpuUsagePercent = (initialCpuUsage + finalCpuUsage) / 2;
    experiment.AvailableMemoryMB = availableMemoryMB;
}
```

Рисунок 3.13 – Код методу «MeasureLinkedListOperations»

Після цього ініціалізується пов'язаний список, у який додаються елементи на основі кількості операцій, визначених у експерименті. Виконується доступ до

першого елемента, видаляється останній елемент і список очищується. Після зупинки секундоміра збираються фінальні показники пам'яті та завантаження процесора. Усі отримані дані – час виконання, обсяг використаної пам'яті та середнє завантаження процесора – зберігаються у відповідному об'єкті експерименту для подальшого аналізу.

Метод на рис. 3.14 відповідає за вимірювання ефективності операцій, пов'язаних із фрагментацією пам'яті. Спочатку виконуються початкові виміри використаної пам'яті та завантаження процесора. Для створення фрагментованих блоків пам'яті використовується список, куди додаються масиви байтів різного розміру. Розмір кожного блоку визначається випадковим чином у межах заданого діапазону, після чого кожен масив заповнюється даними. Щоб симулювати фрагментацію, частина блоків випадково видаляється з пам'яті, і викликається примусова збірка сміття для очищення пам'яті.

```
private void MeasureMemoryFragmentation(MemoryFragmentation experiment) {
    PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time", "_Total");
    PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    // Початкові виміри пам'яті та ЦП
    long initialMemory = GC.GetTotalMemory(true);
    double initialCpuUsage = cpuCounter.NextValue();
    // Список для зберігання фрагментованих блоків
    List<byte[]> fragmentedBlocks = new List<byte[]>();
    Random random = new Random();
    // Створення фрагментованої пам'яті
    for (int i = 0; i < experiment.NumFragments; i++) {
        // Визначення мінімального розміру блоку для запобігання помилкам
        int fragmentSize = experiment.FragmentSize < 1024 ? random.Next(1, experiment.FragmentSize)
            : random.Next(1024, experiment.FragmentSize);
        byte[] block = new byte[fragmentSize];
        // Ініціалізація масиву
        for (int j = 0; j < block.Length; j++) {
            block[j] = (byte)(j % 256);
        }
        // Додаємо блок у список
        fragmentedBlocks.Add(block);
        // Створення фрагментації шляхом випадкового звільнення частини блоків
        if (i % 3 == 0 && fragmentedBlocks.Count > 0) {
            fragmentedBlocks.RemoveAt(random.Next(fragmentedBlocks.Count)); // Видаляємо випадковий блок
            GC.Collect(); // Примусова збірка сміття
        }
    // Вивільнення пам'яті
    fragmentedBlocks.Clear();
    GC.Collect(); // Примусове очищення пам'яті
}
```

Рисунок 3.14 – Фрагмент коду «MeasureMemoryFragmentation»

Після завершення цього процесу секундомір зупиняється, і виконуються фінальні виміри використаної пам'яті та процесора. Результати експерименту, включаючи час виконання, обсяг використаної пам'яті та середнє навантаження на процесор, зберігаються в об'єкті експерименту. Після цього звільняється пам'ять, і ще раз викликається збірка сміття для повного очищення фрагментованих блоків.

На рис. 3.15 метод призначений для вимірювання ефективності виділення пам'яті на великій купі об'єктів. На початку ініціалізуються лічильники для відстеження завантаження процесора та доступної пам'яті, а також запускається секундомір для вимірювання часу операцій. Далі створюється список для збереження великих об'єктів, де для кожної операції виділяється пам'ять під масив байтів певного розміру. Кожен масив заповнюється даними, і цей масив додається до списку для подальшого зберігання.

```
private void MeasureLargeObjectAllocation(LargeObject experiment) {
    PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time", "_Total");
    PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    long initialMemory = GC.GetTotalMemory(true);
    double initialCpuUsage = cpuCounter.NextValue();
    // Список для великих об'єктів
    List<byte[]> largeObjects = new List<byte[]>();
    // Виділення пам'яті на великий масив байтів
    for (int i = 0; i < experiment.NumAllocations; i++) {
        // Виділення пам'яті для масиву байтів
        byte[] largeArray = new byte[experiment.ArraySize];
        // Ініціалізація масиву
        for (int j = 0; j < largeArray.Length; j++) {
            largeArray[j] = (byte)(j % 256); // Заповнення масиву байтами
        }
        // Додаємо масив до списку для збереження
        largeObjects.Add(largeArray);
    }
    stopwatch.Stop();
    long finalMemory = GC.GetTotalMemory(false);
    double finalCpuUsage = cpuCounter.NextValue();
    double availableMemoryMB = ramCounter.NextValue();
    experiment.ExecutionTimeMilliseconds = stopwatch.Elapsed.TotalMilliseconds;
    experiment.MemoryUsedMB = (finalMemory - initialMemory) / (1024 * 1024);
    experiment.AverageCpuUsagePercent = (initialCpuUsage + finalCpuUsage) / 2;
    experiment.AvailableMemoryMB = availableMemoryMB;
}
```

Рисунок 3.15 – Код методу «MeasureLargeObjectAllocation»

Після того як всі масиви створені та ініціалізовані, секундомір зупиняється. Далі отримуються кінцеві показники використаної пам'яті та процесора, щоб оцінити загальну продуктивність операцій. Результати експерименту, такі як час

виконання, використана пам'ять та середнє навантаження на процесор, зберігаються в об'єкті LargeObject, що дозволяє аналізувати ефективність цих операцій у подальших дослідженнях.

Метод на рис. 3.16 використовується для вимірювання ефективності операцій зі стрічками. Спочатку здійснюються початкові виміри використання пам'яті та завантаження процесора, після чого розпочинається відлік часу. У ході виконання операцій система оперує рядком, на основі якого генеруються нові варіації шляхом зміни символів у рядку. Для кожного символу у вхідному рядку виконується циклічне шифрування символів, що генерує нові рядки, які зберігаються у списку. Кількість варіацій залежить від довжини рядка та кількості операцій, що викликає поступове збільшення обсягу використаної пам'яті та тривалості операцій.

```
private void MeasureStringManipulation(StringManipulations experiment) {
    PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time", "_Total");
    PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    long initialMemory = GC.GetTotalMemory(true);
    double initialCpuUsage = cpuCounter.NextValue();
    // Операції з рядками
    string result = experiment.EnterString;
    List<string> generatedStrings = new List<string>(); // Збереження нових рядків
    // Для кожного символу генеруємо кілька нових рядків, змінюючи символ
    for (int i = 0; i < experiment.EnterString.Length; i++) {
        char[] charArray = result.ToCharArray();
        // Створюємо нову варіацію, змінюючи символ на індексі
        for (int j = 0; j < experiment.OperationCount; j++) {
            // Генеруємо новий символ на базі існуючого
            charArray[i] = (char)((charArray[i] + j) % 256); // Шифрування через циклічну зміну ASCII

            // Створюємо новий рядок і додаємо до списку
            string newString = new string(charArray);
            generatedStrings.Add(newString);
        }
    }
```

Рисунок 3.16 – Фрагмент коду «MeasureStringManipulation»

Після завершення цих операцій секундомір зупиняється, і отримуються фінальні показники використаної пам'яті та завантаження процесора. Результати експерименту, такі як час виконання, обсяг використаної пам'яті та середнє навантаження на процесор, зберігаються в об'єкті експерименту для подальшого аналізу. Потім ці результати відображаються у текстовому полі LogsTBox, що дозволяє користувачу переглянути деталі експерименту в реальному часі.

Рис. 3.17 відображає код методу, який відповідає за вимірювання ефективності операцій зі словниками. На початку проводяться початкові виміри використання пам'яті та завантаження процесора, а також запускається секундомір для відстеження часу виконання операцій. Потім створюється словник, у який додаються елементи відповідно до кількості операцій, зазначених у параметрах експерименту. Після цього метод перевіряє роботу зі словником, зокрема доступ до першого елемента, видалення останнього елемента і повне очищення словника.

```
private void MeasureDictionarieAllocation(DictionarieOperation experiment) {
  PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time", "_Total");
  PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
  Stopwatch stopwatch = new Stopwatch();
  stopwatch.Start();
  long initialMemory = GC.GetTotalMemory(true);
  double initialCpuUsage = cpuCounter.NextValue();
  // Ініціалізація словника
  Dictionary<int, int> dictionary = new Dictionary<int, int>();
  // Додавання елементів у словник
  for (int i = 0; i < experiment.OperationCount; i++) {
    dictionary[i] = i; // Додаємо елемент у словник
  }
  // Операції з елементами словника: доступ до елементів і видалення
  if (dictionary.Count > 0) {
    // Доступ до першого елемента
    int firstKey = dictionary.Keys.First();
    int firstValue = dictionary[firstKey];
    // Видалення останнього елемента
    int lastKey = dictionary.Keys.Last();
    dictionary.Remove(lastKey);
    // Очищення словника
    dictionary.Clear();
  }
```

Рисунок 3.17 – Фрагмент коду «MeasureDictionarieAllocation»

Після виконання всіх операцій секундомір зупиняється, і знову фіксуються фінальні значення використаної пам'яті та завантаження процесора. Усі отримані результати – час виконання, обсяг використаної пам'яті та середнє навантаження на процесор – зберігаються у відповідному об'єкті експерименту, що дозволяє в подальшому аналізувати ефективність цих операцій.

3.2 Тестування та налагодження програмного забезпечення

Під час процесу розробки програмного забезпечення одним із важливих етапів є його тестування та налагодження. Ці дії дозволяють забезпечити стабільність роботи системи, перевірити правильність виконання всіх функцій та виявити

можливі недоліки в логіці або продуктивності. Для програм, що досліджують часову ефективність різних операцій, таких як виділення та ініціалізація масивів, особливо важливо переконатися, що всі експерименти виконуються коректно, а отримані результати є достовірними.

На початку було проведено тестування форми «Виділення та ініціалізація масивів». Тестування охоплювало різні сценарії з метою перевірки правильності введення даних, обчислення ефективності виділення пам’яті та коректного відображення результатів експерименту у реальному часі. Система також відображала результати експериментів у режимі реального часу для аналізу, що дозволяло виявити закономірності у використанні ресурсів. Результати тестування представлено на рис. 3.18, що демонструє залежність часу виконання операцій і використання пам'яті від кількості масивів і розмірів їх елементів.

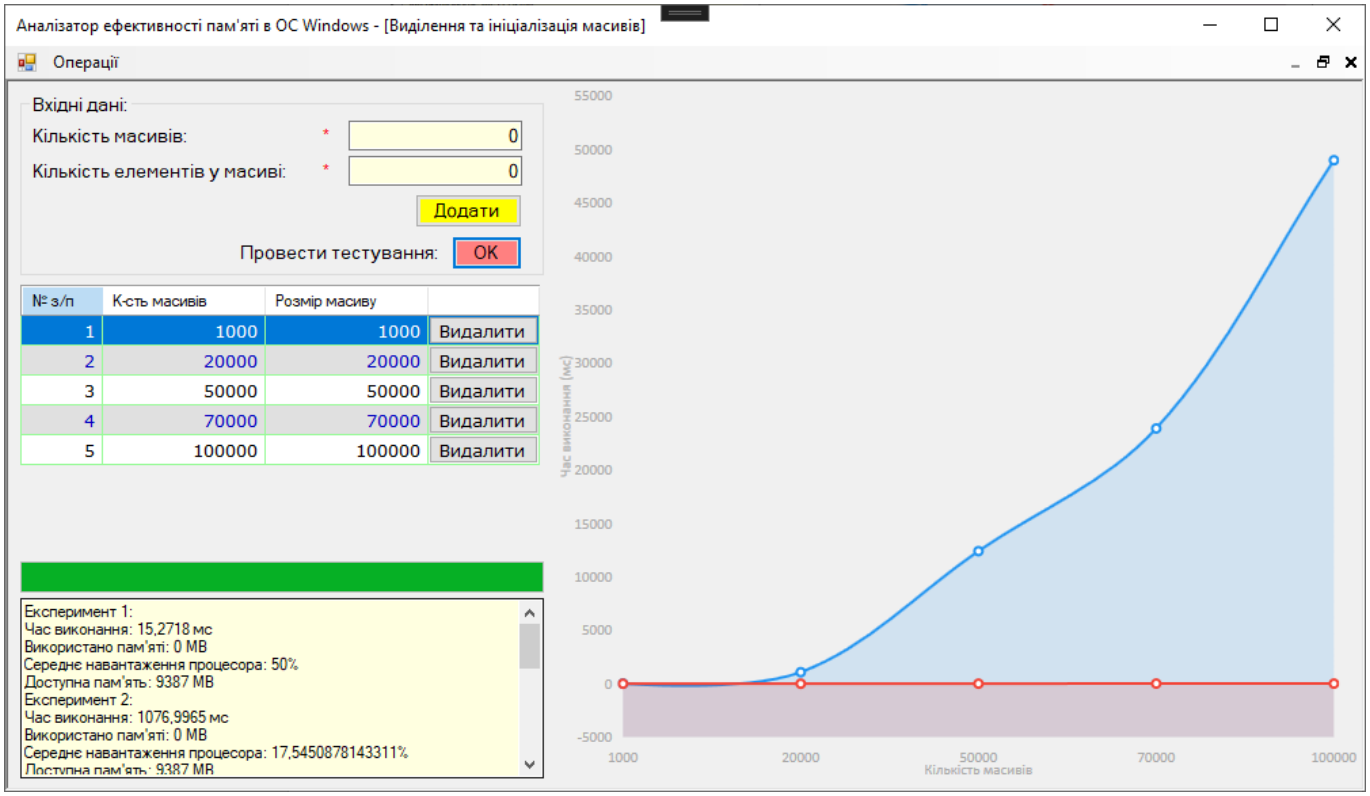

Рисунок 3.18 – Результати тестування форми «Виділення та ініціалізація масивів»

Тестування форми «Операції з динамічними структурами» дозволило перевірити працездатність програмного забезпечення під час виконання ключових операцій, таких як додавання елементів, доступ до них, видалення, а також коректність відображення даних на графіку. У ході кожного тестування результати

відображалися в реальному часі, включаючи інформацію про використання пам'яті, час виконання операцій та навантаження на процесор. На рис. 3.19 наведено графічне представлення результатів тестування. Перевірка коректного функціонування форми також охоплювала можливість безперервної роботи з великою кількістю елементів у структурі та відображення графічних даних без помилок. Це дозволило оцінити стійкість і надійність системи при роботі з динамічними списками.

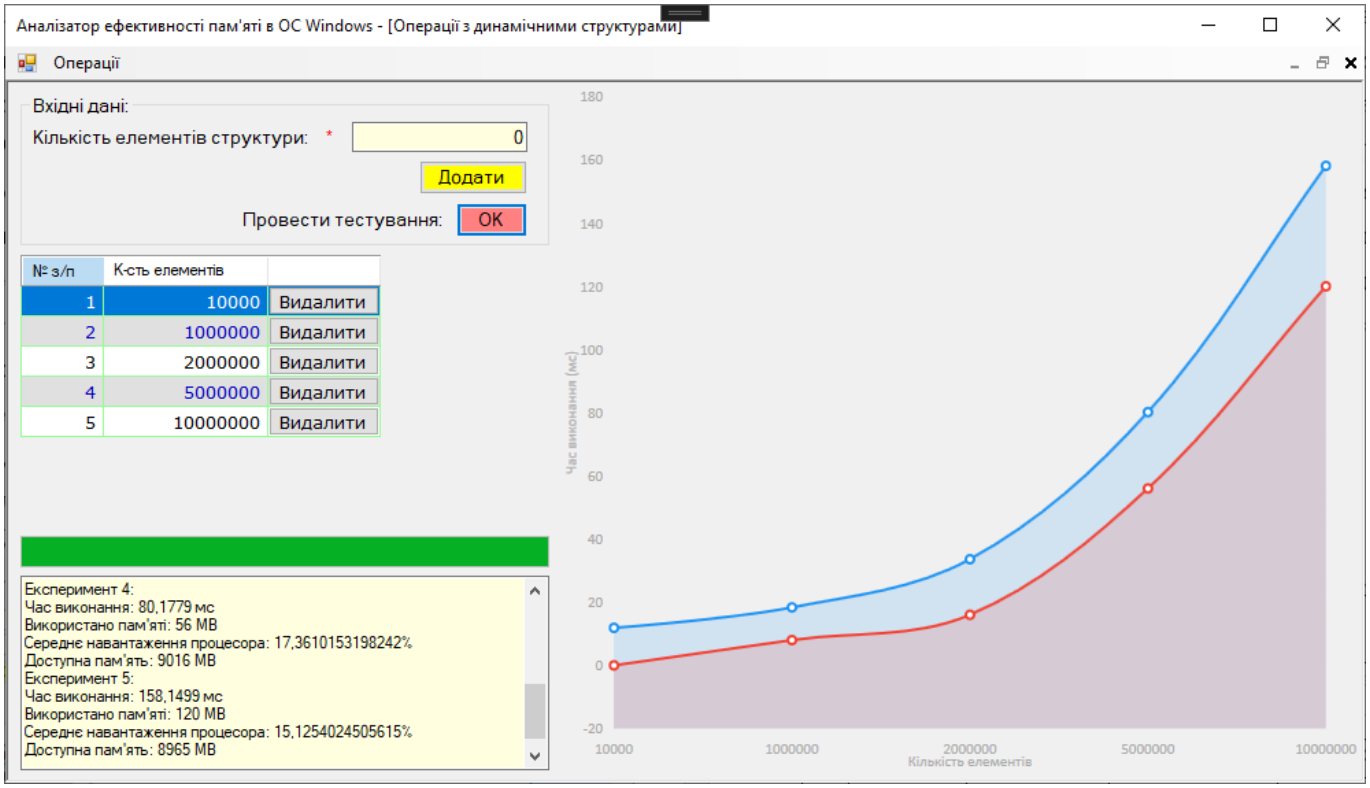

Рисунок 3.19 – Результати тестування форми «Операції з динамічними структурами»

Тестування форми «Операції з пов'язаним списком» дозволило перевірити працездатність системи під час виконання операцій додавання елементів, доступу до них, видалення, а також коректного відображення графіку результатів. Кожен експеримент супроводжувався виведенням показників у реальному часі, які включали використання пам'яті, час виконання операцій і навантаження на процесор. Результати тестування, представлені на рис. 3.20, підтверджують стабільність роботи системи та її здатність адекватно виконувати маніпуляції з великими обсягами даних у динамічних структурах. Важливою складовою

перевірки було оцінювання відображення даних на графіку, що дозволило виявити можливі проблеми у візуалізації та загальній стійкості програмного забезпечення.

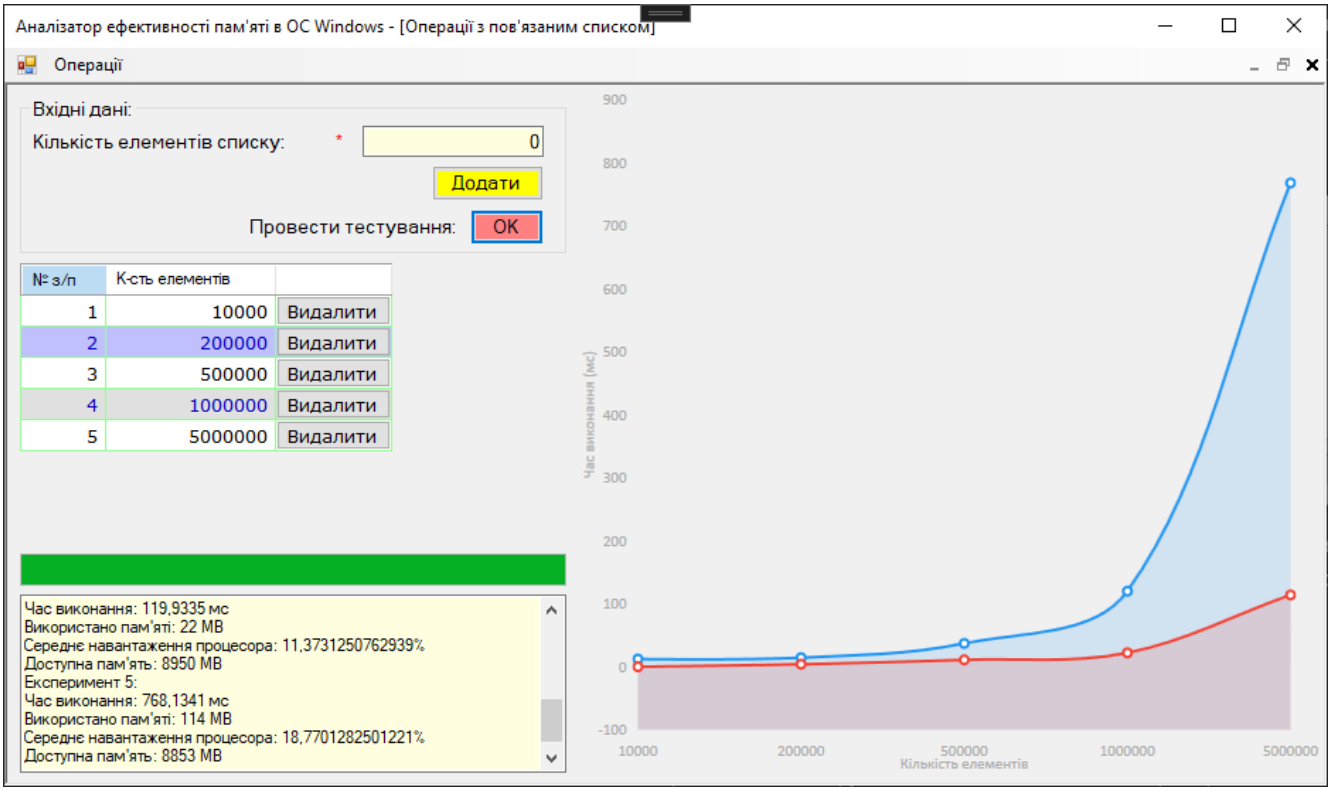

Рисунок 3.20 – Результати тестування форми «Операції з пов'язаним списком»

Тестування форми «Робота зі словниками» дало змогу перевірити працездатність системи під час виконання ключових операцій зі словниками, таких як додавання елементів, доступ до них і видалення з колекції. Під час кожного експерименту результати відображалися в реальному часі, що включало показники використання пам'яті, час виконання операцій та навантаження на процесор. Як видно з графіка на рис. 3.21, система коректно обробляла операції зі словниками, забезпечуючи належну продуктивність і стійкість. Важливою частиною перевірки була оцінка точності відображення результатів на графіку та стабільності роботи під час маніпуляцій з великою кількістю елементів у словнику, що підтвердило надійність програмного забезпечення.

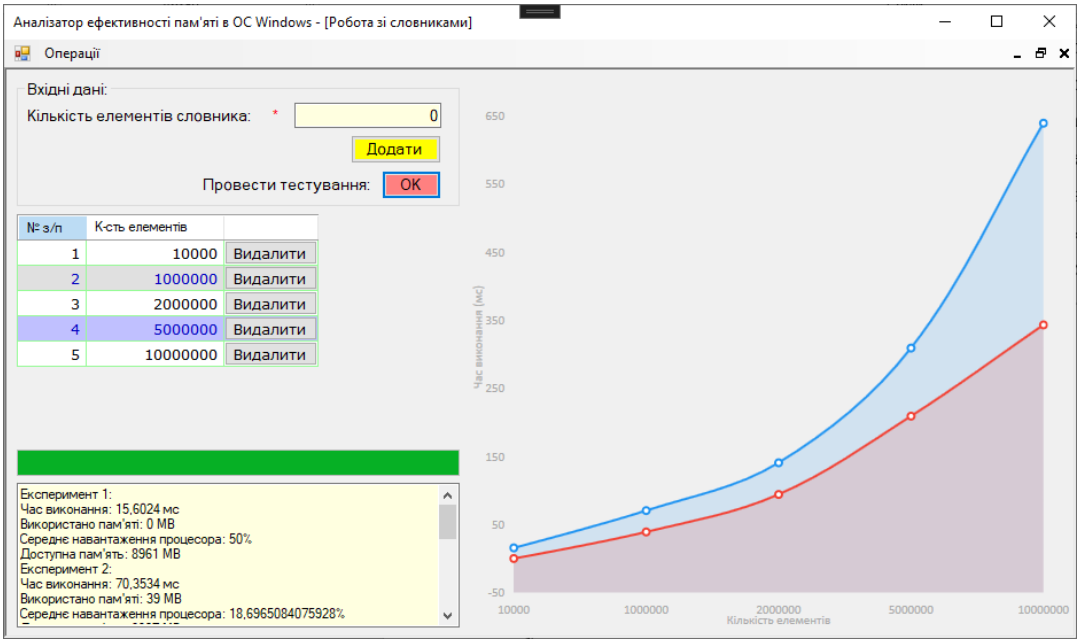

Рисунок 3.21 – Результати тестування форми «Робота із словниками»

Тестування форми «Маніпуляції зі стрічками» було спрямоване на перевірку працездатності системи під час виконання операцій зі строковими даними. До операцій належали зміна символів у тексті, створення нових рядків і виконання різних маніпуляцій над ними. Усі результати експериментів відображалися в реальному часі, що включало показники часу виконання операцій та використання системної пам'яті. На рис. 3.22 наведено результати тестування, де можна побачити залежність між кількістю операцій та використанням ресурсів системи.

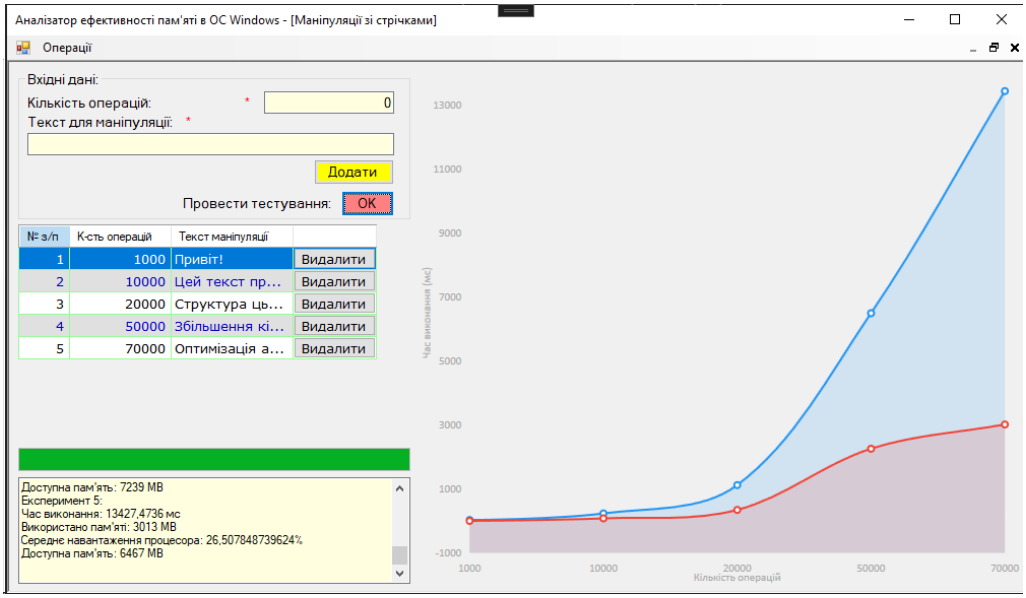

Рисунок 3.22 – Результати тестування форми «Маніпуляції зі стрічками»

Тестування форми для аналізу працездатності системи при виділенні пам’яті на великій кількості об'єктів продемонструвало коректність виконання операцій за різної кількості виділень та об'єктів (рис. 3.23). Під час тестування було перевірено, як збільшення кількості об'єктів та виділень впливає на час виконання операцій та використання системних ресурсів. Усі результати відображалися в реальному часі, зокрема використання пам'яті, навантаження на процесор та час виконання операцій. Графік чітко показує залежність між кількістю виділень і навантаженням на ресурси, що дозволяє зробити висновки про стійкість та продуктивність програми під час роботи з великими обсягами даних.

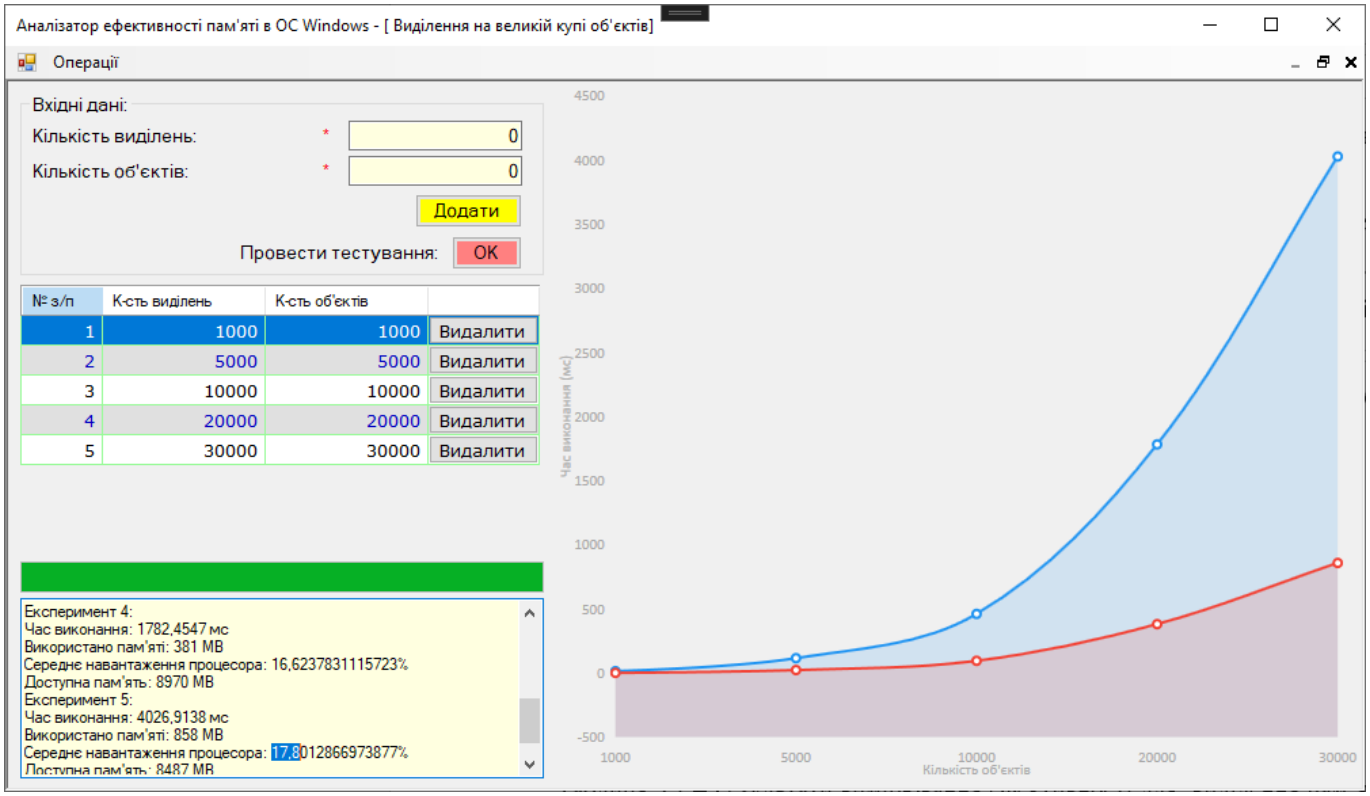

Рисунок 3.23 – Результати тестування форми «Виділення на великій купі об’єктів»

Тестування форми «Фрагментація пам'яті» дозволило оцінити працездатність системи під час роботи з різною кількістю фрагментів та максимальними розмірами блоків. Результати експериментів показали, що зі збільшенням кількості фрагментів значно зростає час виконання операцій. Графічне представлення на рис. 3.24 наочно демонструє взаємозв'язок між кількістю фрагментів та використанням системних ресурсів, таких як пам'ять і процесорний час. Відображення результатів у реальному часі дозволило виявити закономірності, що виникають при обробці великих обсягів

фрагментованих даних. Збільшення кількості блоків призводило до значного зростання навантаження на ресурси, що підтверджує необхідність ефективного управління пам'яттю для забезпечення стабільної продуктивності системи.

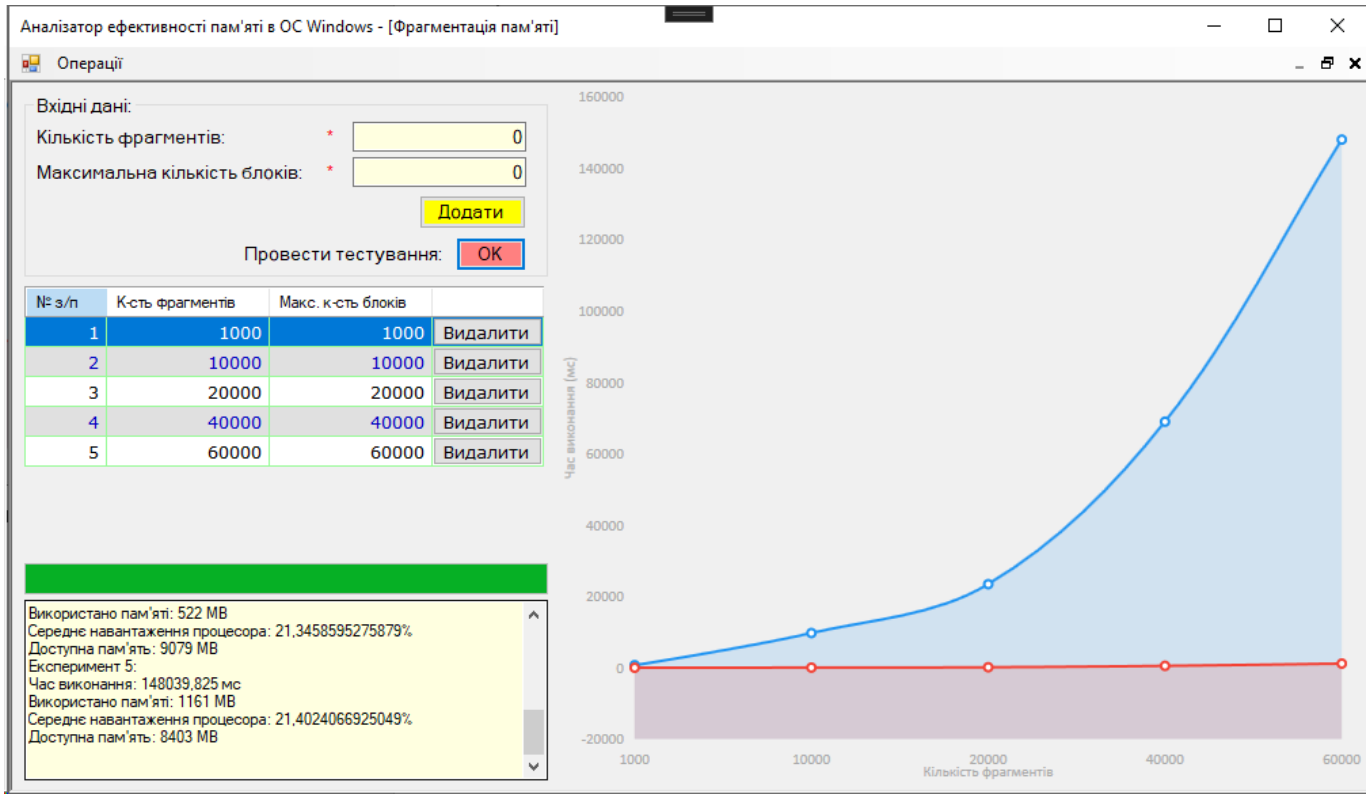

Рисунок 3.24 – Результати тестування форми «Фрагментація пам’яті»

Тестування форми для аналізу ефективності операцій з небезпечним кодом і покажчиками дозволило оцінити залежність часу виконання від кількості операцій. Як показано на рис. 3.25, зростання кількості операцій призводить до збільшення часу їх виконання, що очікувано для роботи з покажчиками, оскільки небезпечний код вимагає додаткового контролю за доступом до пам'яті. Результати також демонструють, що використання пам'яті зростає не так суттєво, як процесорне навантаження, що вказує на основний вплив операцій з покажчиками на обчислювальні ресурси. Кожен експеримент показав важливість моніторингу при роботі з небезпечним кодом, особливо коли обробляються великі обсяги даних, для забезпечення надійної і стабільної роботи системи.

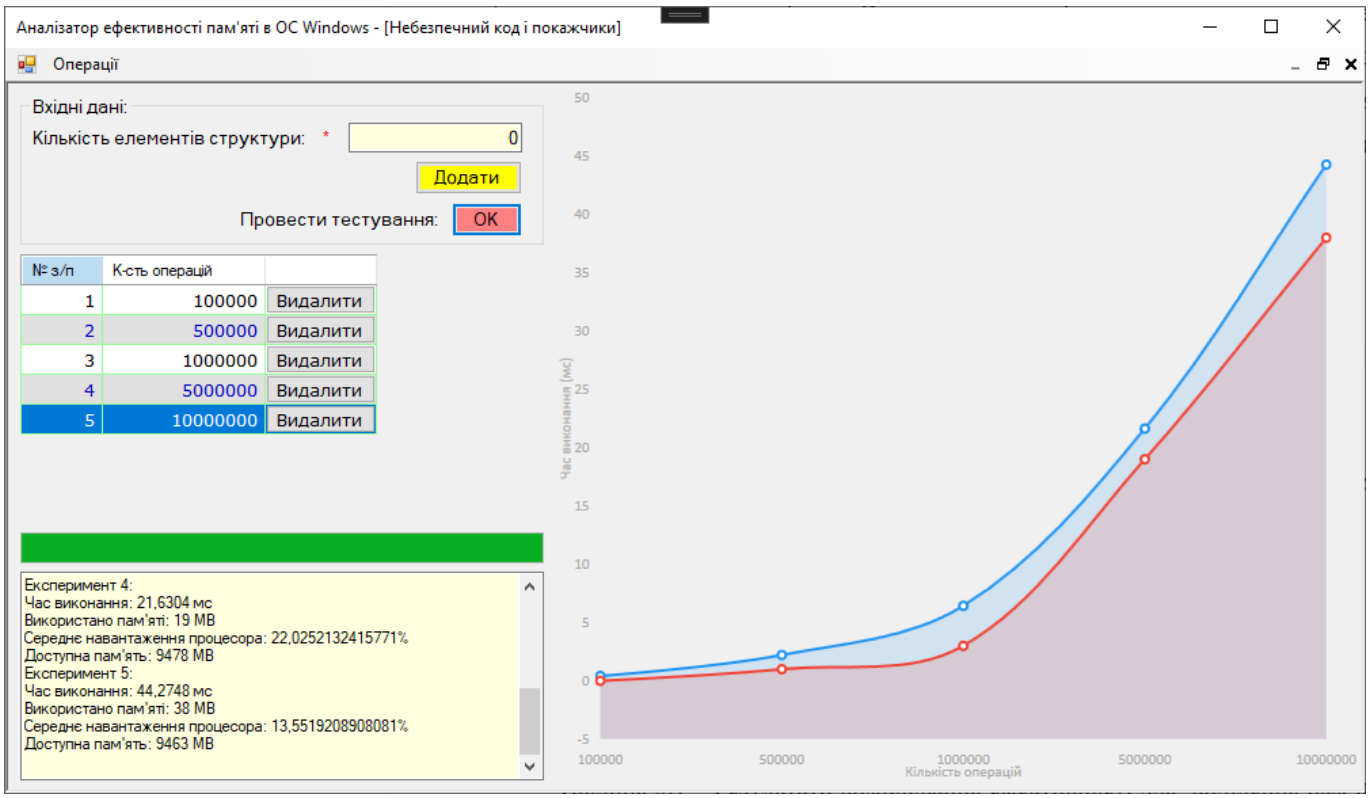

Рисунок 3.25 – Результати тестування форми «Небезпечний код та покажчики»

Під час тестування та налагодження програмного забезпечення особлива увага була приділена працездатності програми, її стійкості до помилок та коректності виконання усіх основних функцій. Тестування охоплювало різні аспекти, такі як введення даних користувачем, виконання експериментів з різними обсягами операцій, а також правильне відображення результатів. Програма пройшла всі етапи тестування без критичних помилок, що свідчить про її стабільність та надійність. Під час налагодження було усунуто незначні недоліки в роботі інтерфейсу, а також оптимізовано алгоритми для забезпечення швидкої реакції на дії користувача та точного вимірювання параметрів системи.

3.3 Аналіз отриманих результатів

Аналіз результатів експериментів дозволяє оцінити ефективність розробленого програмного забезпечення з точки зору використання системних ресурсів та часу виконання операцій. У процесі тестування було зібрано дані про продуктивність програми при роботі з різними обсягами даних та кількістю

операцій, що дозволяє виявити ключові закономірності у використанні пам'яті, навантаженні на процесор і часу виконання.

На початку проведено дослідження з метою вимірювання ефективності роботи системи при виконанні операцій з різними масивами. Результати експериментів, які відображають час виконання, використання пам'яті та середнє навантаження на процесор, занесені у табл. 3.1. Дані досліджень демонструють, що зі збільшенням розміру масивів значно зростає час виконання операцій, при цьому використання пам'яті залишається мінімальним для невеликих масивів, але зростає для більших обсягів даних.

Таблиця 3.1 – Результати вимірювання ефективності для виділення пам’яті та ініціалізації масивів

№ з/п	Кількість масивів	Розмір масиву	Час виконання (мс.)	Використано пам'яті (Мб.)	Середнє навантаження процесора (%)
1	1000	1000	15,27	0	50
2	20000	20000	1076,99	0	17,54
3	50000	50000	12403,32	5	25,09
4	70000	70000	23877,37	12	25,21
5	100000	100000	48972,34	10	27,03

У першому випадку, для масиву розміром 1000 елементів, час виконання операції становить 15,27 мс, при цьому пам'ять не використовується, а навантаження на процесор досягає 50%. Однак, зі збільшенням кількості масивів і їх розміру до 100 000 елементів, час виконання збільшується до 48972,34 мс, а використання пам'яті досягає 10 Мб, при цьому середнє навантаження на процесор поступово зменшується до 27,03%.

Згідно з отриманими даними, графік часу виконання операцій та використання пам'яті в залежності від розміру масиву демонструє явне зростання із збільшенням кількості елементів у масиві. Також видно, що навантаження на процесор спадає при роботі з більшими масивами, але все ще залишається значним.

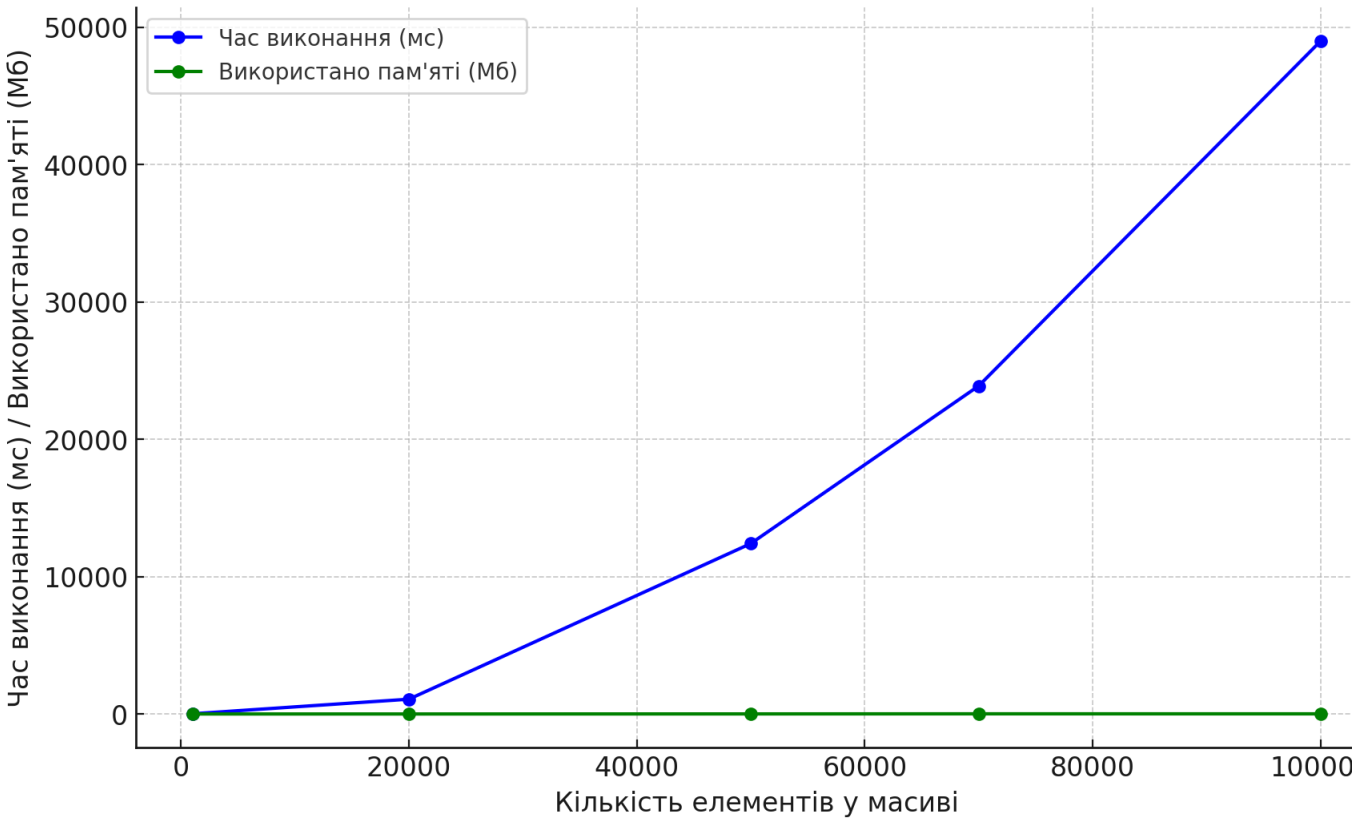

Рисунок 3.26 – Залежність часу виконання та використаної пам'яті від розміру масивів

На графіку наведено результати тестування часу виконання операцій та використання пам'яті залежно від розміру масивів. Як видно з графіка, час виконання операцій суттєво збільшується зі зростанням кількості елементів у масиві, що є очікуваним результатом для великих обсягів даних. Використання пам'яті демонструє незначні зміни, досягаючи свого максимуму при великих розмірах масивів.

Результати підтверджують тенденцію до зростання часу виконання операцій зі збільшенням кількості елементів, що вказує на потребу в оптимізації при роботі з великими масивами даних для підвищення продуктивності системи.

Було проведено дослідження для оцінки ефективності роботи з динамічними структурами, результати якого занесено до табл. 3.2. Дані свідчать про зростання часу виконання операцій та використання пам'яті зі збільшенням кількості елементів у структурі. Наприклад, для кількості елементів 10 000 час виконання становив лише 11,88 мс, а використання пам'яті – 0 Мб. Проте для структури з 10 мільйонами елементів час виконання зріс до 158,14 мс, а використання пам'яті досягло 120 Мб.

Таблиця 3.2 – Результати вимірювання ефективності для операцій з динамічними структурами

№ з/п	Кількість елементів структури	Час виконання (мс.)	Використано пам'яті (Мб.)	Середнє навантаження процесора (%)
1	10000	11,88	0	0
2	1000000	18,35	8	50
3	2000000	33,6	16	10,83
4	5000000	80,17	56	17,36
5	10000000	158,14	120	15,12

На графіку показано залежність часу виконання та використання пам'яті від кількості елементів у динамічній структурі (рис. 3.27).

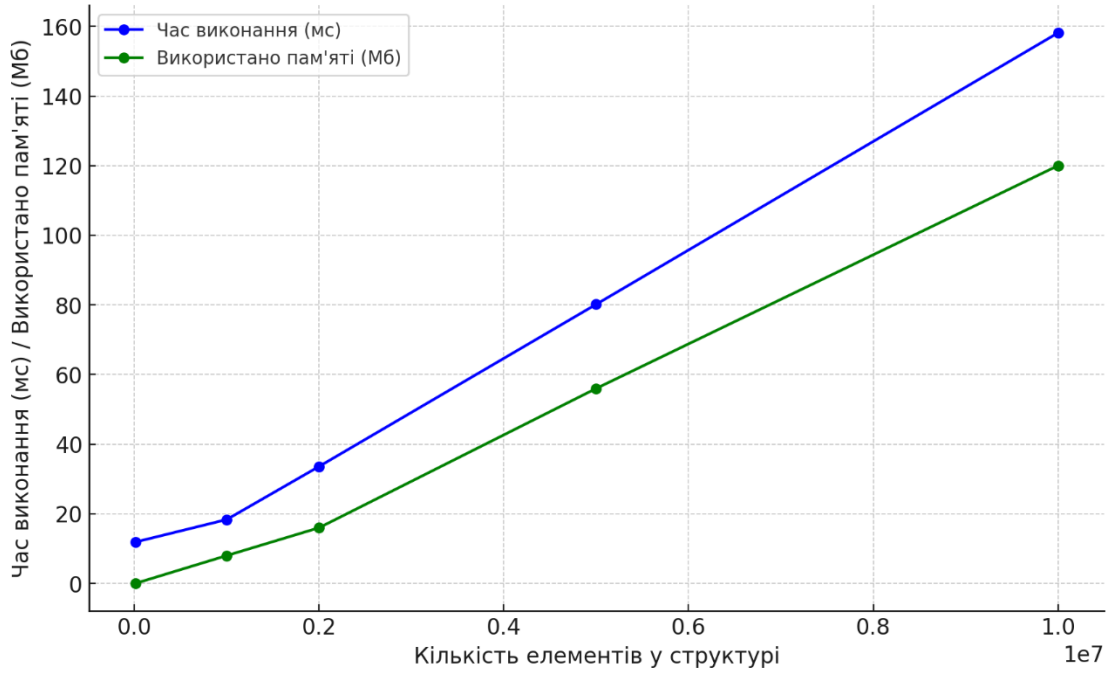

Рисунок 3.27 – Залежність часу виконання та використаної пам'яті від кількості елементів у динамічній структурі

Загалом спостерігається чітке зростання обох показників у міру збільшення розміру структури, що вказує на закономірність збільшення навантаження на систему. Це підтверджує необхідність оптимізації при роботі з великими динамічними структурами для збереження стабільної продуктивності.

Наступним кроком проведено дослідження для оцінки ефективності операцій з пов'язаним списком. Результати показують, що зі збільшенням кількості елементів у

списку значно зростає час виконання операцій і використання пам'яті. Для списку з 10 000 елементів час виконання становив лише 12,28 мс, тоді як для 5 мільйонів елементів він збільшився до 768,13 мс. Використання пам'яті також зростає з 0 Мб до 114 Мб для найбільших списків.

Результати експериментів занесені в табл. 3.3 та показані на графіку (рис. 2.28), де можна побачити чітку залежність часу виконання та використання пам'яті від кількості елементів у пов'язаному списку.

Таблиця 3.3 – Результати вимірювання ефективності для операцій з пов'язаним списком

№ з/п	Кількість елементів списку	Час виконання (мс.)	Використано пам'яті (Мб.)	Середнє навантаження процесора (%)
1	10000	12,28	0	0
2	200000	14,42	4	50
3	500000	37,06	11	3,01
4	1000000	119,93	22	11,37
5	5000000	768,13	114	18,77

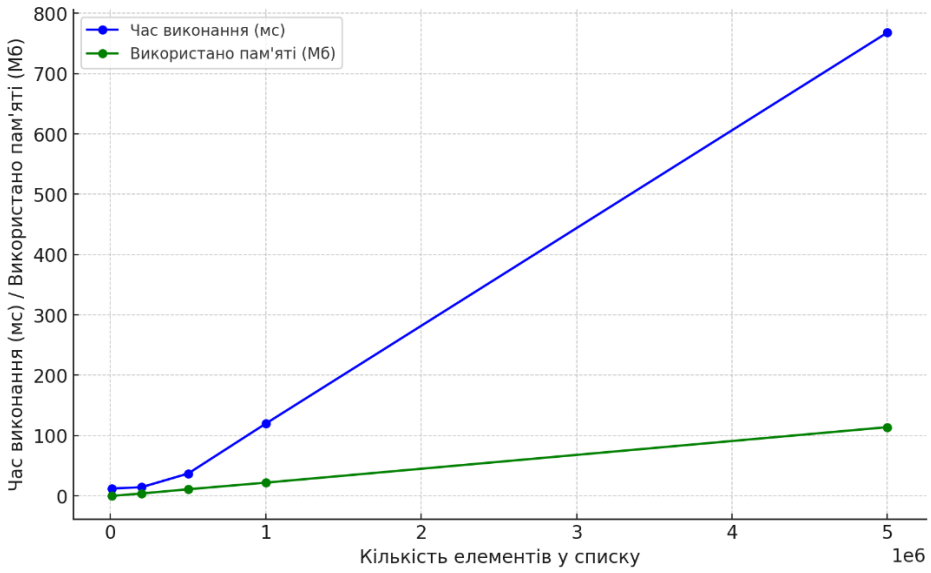

Рисунок 3.28 – Залежність часу виконання та використаної пам'яті від кількості елементів у пов'язаному списку

Отже, при збільшенні кількості елементів у структурі час виконання і використання пам'яті суттєво зростають, що вказує на потребу в ефективних методах роботи з великими обсягами даних у пов'язаних списках.

Було проведено дослідження для оцінки ефективності операцій зі словниками, результати якого показують значне зростання часу виконання та використання пам'яті зі збільшенням кількості елементів. Результати тестування занесені до табл. 3.4.

Таблиця 3.4 – Результати вимірювання ефективності для операцій із словниками

№ з/п	Кількість елементів словника	Час виконання (мс.)	Використано пам'яті (Мб.)	Середнє навантаження процесора (%)
1	10000	15,6	0	50
2	1000000	70,35	39	18,69
3	2000000	140,53	94	16,25
4	5000000	308,98	209	24,06
5	10000000	639,17	343	16,55

Графік на рис. 2.29 демонструє закономірне зростання використання системних ресурсів при збільшенні обсягу даних. Час виконання операцій та використання пам'яті зростають прямо пропорційно до кількості елементів у словнику, що підтверджується графіком.

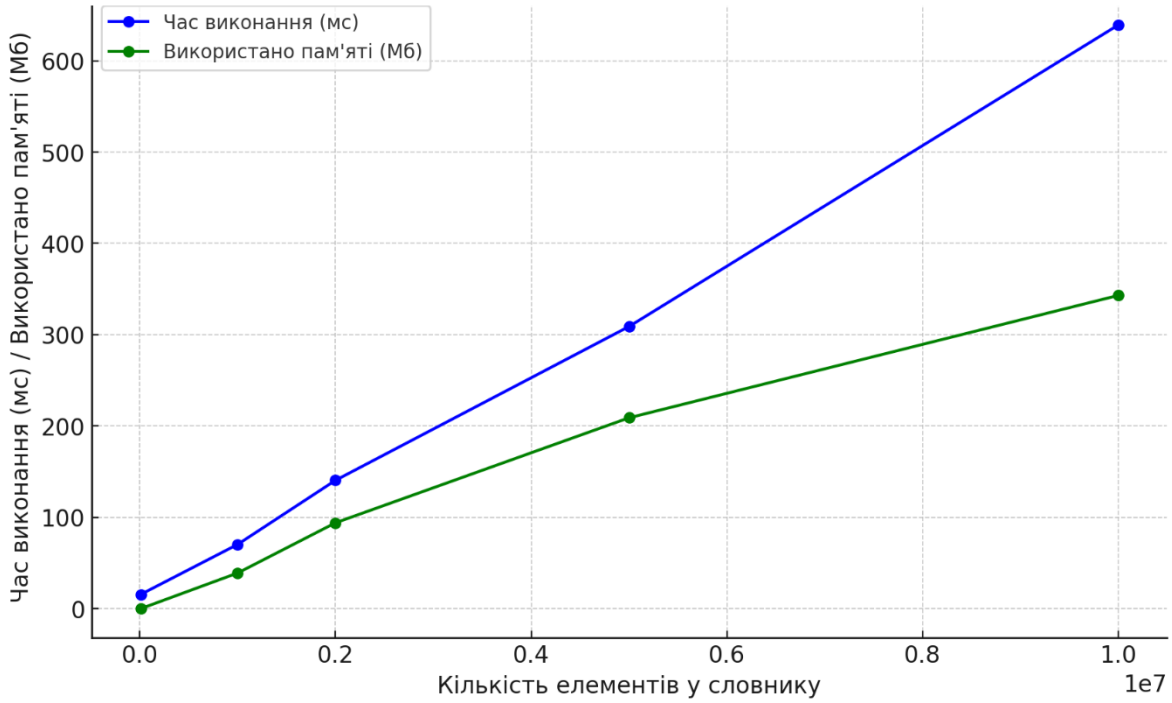

Рисунок 3.29 – Залежність часу виконання та використаної пам'яті від кількості елементів у пов'язаному списку

Можна зробити висновок, що зі збільшенням кількості елементів значно зростає як час виконання, так і використання пам'яті, що свідчить про необхідність оптимізації при роботі з великими наборами даних у словниках.

Також, проведено дослідження для оцінки ефективності маніпуляцій із текстовими даними. Збільшення кількості операцій призвело до суттєвого зростання часу виконання та використання пам'яті. Наприклад, для 1000 операцій час виконання становив 29,69 мс, а для 70 000 операцій – 13 427,47 мс, що свідчить про значне навантаження на систему при роботі з великими обсягами тексту. Результати експериментів занесено в табл. 3.5 та відображено на графіку (рис. 3.30). Дані демонструють явну залежність між кількістю операцій та використанням ресурсів, що особливо помітно при великих обсягах маніпуляцій із текстом.

Таблиця 3.5 – Результати вимірювання ефективності для маніпуляцій із стрічками

№ з/п	Кількість операцій	Текст для маніпуляцій	Час виконання (мс.)	Використано пам'яті (Мб.)	Середнє навантаження процесора (%)
1	1000	Привіт!	29,69	2	50
2	10000	Цей текст призначений для перевірки маніпуляцій із рядками.	234,86	80	17,89
3	20000	Структура цього повідомлення дозволяє протестувати ефективність алгоритмів обробки тексту.	1118,99	344	19,79
4	50000	Збільшення кількості символів у тексті дозволяє провести детальнішу оцінку ефективності функцій роботи з рядками, зокрема швидкість обробки та пошук.	6488,81	2255	19,48
5	70000	Оптимізація алгоритмів маніпуляції текстом дозволяє значно підвищити продуктивність у системах з великими обсягами даних та складними операціями.	13427,47	3013	26,5

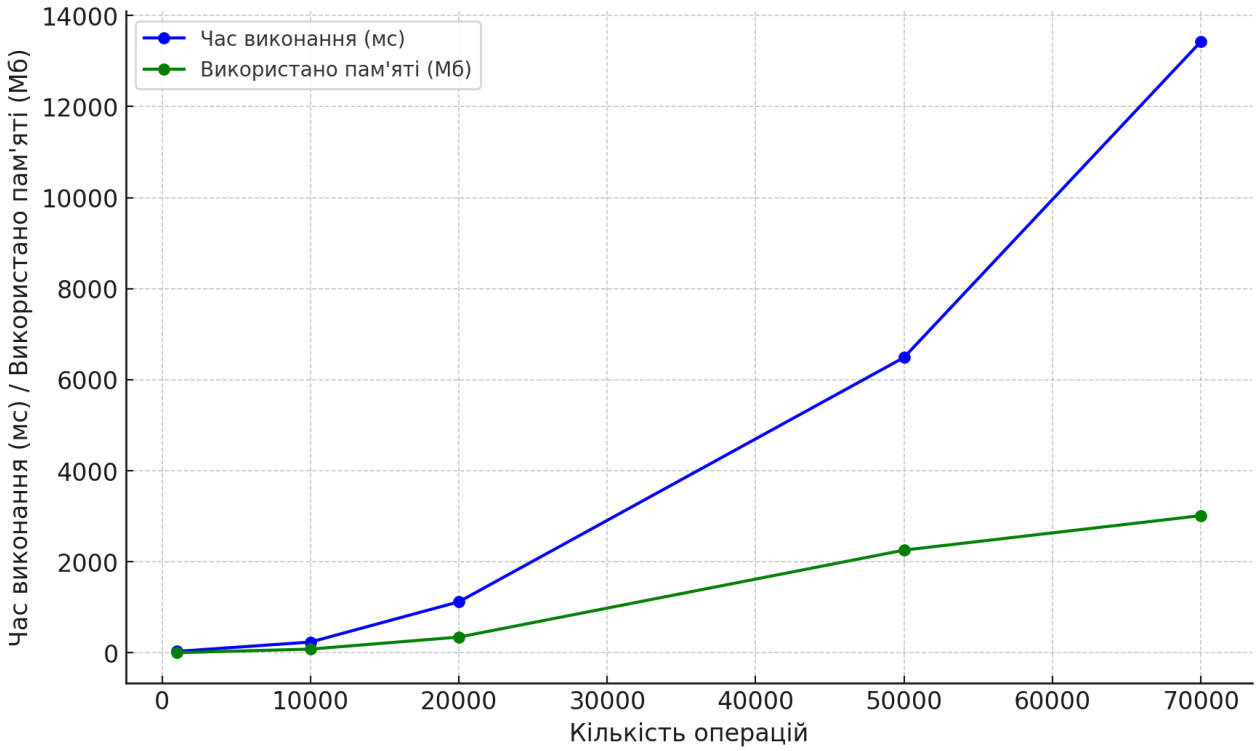

Рисунок 3.30 – Залежність часу виконання та використаної пам'яті від кількості операцій із текстом

Отже, зі збільшенням кількості операцій зростає як час виконання, так і обсяг використаної пам'яті. Це підтверджує необхідність оптимізації для підвищення ефективності обробки текстових даних у системах із великими навантаженнями.

Було проведено дослідження ефективності виділення пам'яті на великій купі об'єктів. Результати показують, що зі збільшенням кількості виділень та об'єктів суттєво зростає як час виконання операцій, так і використання пам'яті. Для 1000 об'єктів час виконання становив 14,89 мс, тоді як для 30 000 об'єктів він збільшився до 4026,91 мс. Використання пам'яті також зросло з 1 Мб до 858 Мб при максимальному навантаженні.

Результати дослідження занесено у табл. 3.6, що відображає залежність часу виконання та використання пам'яті від кількості об'єктів. На графіку, який представлено на рис. 3.31 чітко видно тенденцію до збільшення обох параметрів зі зростанням кількості виділень.

Таблиця 3.6 – Результати вимірювання ефективності для виділення пам’яті на великій купі об'єктів

№ з/п	Кількість виділень	Кількість об’єктів	Час виконання (мс.)	Використано пам'яті (Мб.)	Середнє навантаження процесора (%)
1	1000	1000	14,89	1	50
2	5000	5000	115,8	23	13,76
3	10000	10000	460,58	95	16,30
4	20000	20000	1782,45	381	16,62
5	30000	30000	4026,91	858	17,80

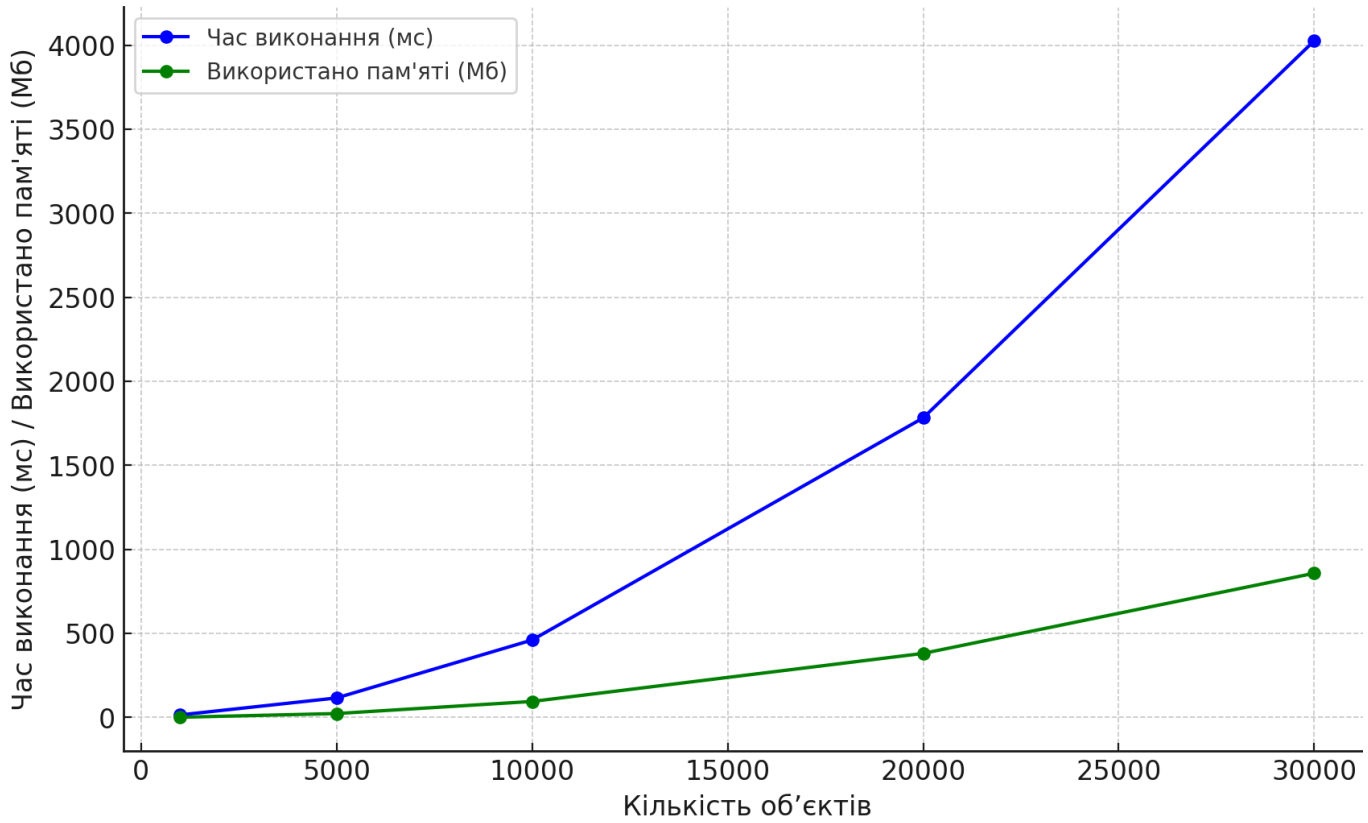

Рисунок 3.31 – Залежність часу виконання та використаної пам'яті від кількості об’єктів

Можна із впевненістю сказати, що при збільшенні кількості об'єктів зростає навантаження на систему, особливо у плані часу виконання, що вимагає оптимізації процесу управління пам'яттю для покращення продуктивності.

Дослідження, результати якого занесено до табл. 3.7, дозволило оцінити вплив кількості фрагментів на продуктивність системи. Дані свідчать про те, що зі збільшенням кількості блоків значно зростає навантаження на систему.

Таблиця 3.7 – Результати вимірювання ефективності для фрагментації пам’яті

№ з/п	Кількість фрагментів	Максимальна кількість блоків	Час виконання (мс.)	Використано пам'яті (Мб.)	Середнє навантаження процесора (%)
1	1000	1000	1091,85	0	22,07
2	10000	10000	12125,68	35	20,55
3	20000	20000	30364,32	134	22,69
4	40000	40000	86815,36	523	22,71
5	60000	60000	166559,56	1166	20,85

Результати вимірювання ефективності фрагментації пам'яті показали, що зі збільшенням кількості фрагментів значно зростає час виконання операцій та використання пам'яті (рис. 3.32). Наприклад, для 1000 фрагментів час виконання становив 1091,85 мс, а використання пам'яті – 0 Мб. Однак для 60 000 фрагментів час виконання зріс до 166 559,56 мс, а використання пам'яті – до 1166 Мб.

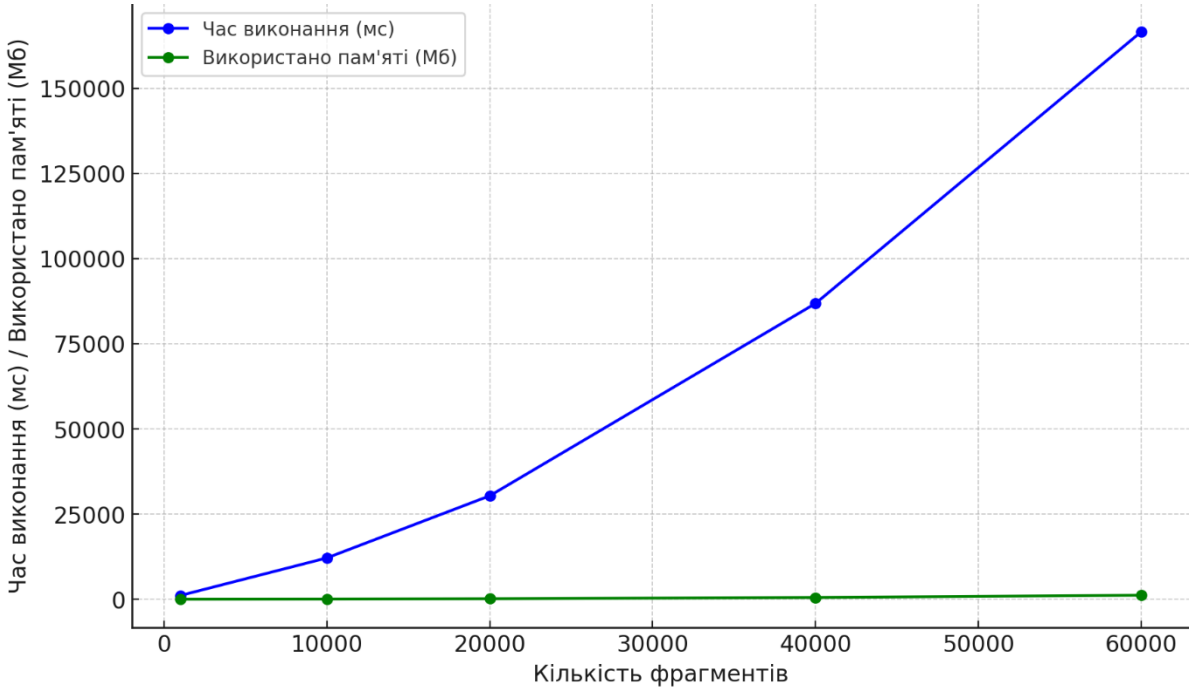

Рисунок 3.32 – Залежність часу виконання та використаної пам'яті від кількості фрагментів

Графік чітко демонструє тенденцію до зростання часу виконання операцій та використання пам'яті зі збільшенням кількості фрагментів. Це вказує на необхідність оптимізації процесів для роботи з великими обсягами фрагментованих даних, щоб уникнути надмірного навантаження на систему.

Дослідження, результати якого занесено до табл. 3.8, дозволило оцінити вплив кількості елементів на продуктивність системи під час використання небезпечного коду та покажчиків. Аналіз даних показує, що зі збільшенням кількості елементів зростає навантаження на систему.

Таблиця 3.8 – Результати вимірювання ефективності з небезпечним кодом та покажчиками

№ з/п	Кількість елементів словника	Час виконання (мс.)	Використано пам'яті (Мб.)	Середнє навантаження процесора (%)
1	100000	0,41	0	50
2	500000	2,21	1	0
3	1000000	6,43	3	50
4	5000000	21,63	19	22,02
5	10000000	44,27	38	13,55

Результати вимірювання ефективності для роботи з небезпечним кодом та покажчиками показали, що зі збільшенням кількості елементів словника зростає як час виконання операцій, так і використання пам'яті (рис. 3.33).

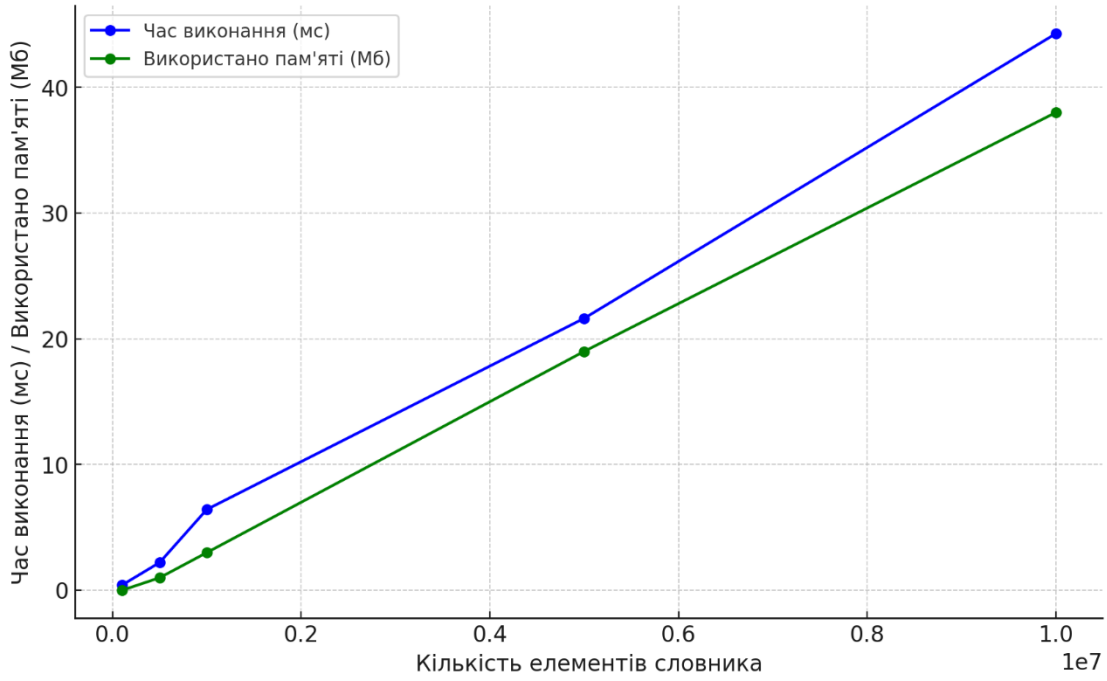

Рисунок 3.33 – Залежність часу виконання та використаної пам'яті від кількості елементів словника (небезпечний код)

На графіку чітко видно залежність між кількістю елементів у словнику та часом виконання операцій, а також використанням пам'яті. Це підкреслює важливість оптимізації під час роботи з великими наборами даних у контексті небезпечного коду.

Проведені експерименти дозволили детально проаналізувати ефективність роботи з динамічною пам'яттю в операційній системі Windows. У результаті тестування було виміряно час виконання операцій, використання пам'яті та середнє навантаження на процесор для різних сценаріїв роботи з динамічними структурами, словниками, небезпечним кодом і покажчиками, а також виділення пам'яті на великій кількості об'єктів. Усі експерименти підтвердили закономірність: зі збільшенням кількості об'єктів, елементів у структурах або фрагментів суттєво зростає час виконання операцій та обсяг використаної пам'яті. Водночас навантаження на процесор залежить від характеру операцій і кількості елементів.

Отримані дані свідчать про те, що для ефективної роботи з великими обсягами даних і динамічними структурами в ОС Windows необхідно враховувати залежність між розміром структур і використанням системних ресурсів. Це дослідження дозволяє виявити слабкі місця в роботі з динамічною пам'яттю і відкриває можливості для подальшої оптимізації алгоритмів і систем управління пам'яттю. Часова ефективність, як показали експерименти, є ключовим фактором у забезпеченні стабільної продуктивності системи під час роботи з великими даними.

ВИСНОВОК

Дана кваліфікаційна робота присвячена розробці та дослідженню системи для оцінки часової ефективності управління динамічною пам'яттю в операційній системі Windows. Основною метою роботи було створення інструменту, який дозволить проводити детальний аналіз операцій з динамічною пам'яттю, включаючи виділення пам'яті, роботу з динамічними структурами, небезпечним кодом та покажчиками, а також маніпуляції зі стрічками та фрагментацію пам'яті.

У першому розділі була розглянута основна концепція часової ефективності, визначені ключові фактори, що впливають на продуктивність роботи з пам'яттю, такі як кількість запитів на виділення пам'яті, ступінь фрагментації, час виконання операцій і використання системних ресурсів. Також проаналізовано методи виділення і звільнення пам'яті в ОС Windows, включаючи механізми роботи з віртуальною пам'яттю та використання HeapAlloc і VirtualAlloc. Окрім цього, проведено огляд існуючих інструментів для тестування пам'яті, таких як Valgrind, HeapMemView і PerfMon, і обґрунтована потреба у розробці нового інструменту.

Другий розділ висвітлює процес вибору інструментів для реалізації системи та проєктування її основних компонентів. Були визначені функціональні вимоги до системи, які охоплювали різні види операцій з пам'яттю, а також розроблені діаграми процесів для кожної ключової операції. На основі аналізу інструментів для розробки було прийнято рішення використовувати мову програмування C# та середовище розробки Visual Studio 2022, що забезпечило оптимальні умови для реалізації та тестування системи.

Третій розділ був присвячений реалізації програмного забезпечення та його тестуванню. Описано створення всіх необхідних модулів, які відповідають за виконання операцій з динамічною пам'яттю, та проведено тестування функціональності системи для кожної з розроблених форм. Тестування дозволило отримати дані щодо ефективності операцій з виділенням пам'яті, динамічними структурами, словниками, маніпуляціями зі стрічками та роботою з небезпечним кодом. Зокрема, було проаналізовано, як збільшення кількості елементів та

фрагментів впливає на час виконання операцій та використання пам'яті, що дало змогу виявити закономірності у навантаженні на систему.

Проведена робота демонструє застосування системного підходу до оцінки часової ефективності операцій з динамічною пам'яттю. Отримані результати вказують на суттєве зростання часу виконання та використання пам'яті зі збільшенням розміру масивів, словників і кількості фрагментів. Це дослідження дає змогу зробити висновок про необхідність оптимізації алгоритмів управління пам'яттю для підвищення продуктивності системи під час роботи з великими обсягами даних. Розроблена система може бути корисною для подальших досліджень у галузі управління пам'яттю в операційних системах та оптимізації процесів виділення і звільнення пам'яті в різних умовах.

ЛІТЕРАТУРА

Літературу оформіть згідно вимог

1. XU Luna. Memtune: Dynamic memory management for in-memory data analytic platforms. In: 2016 IEEE international parallel and distributed processing symposium (IPDPS). IEEE, 2016. 386 p.

2. Jones Richard E. Dynamic memory management: Challenges for today and tomorrow. In: International Lisp Conference. Association of LISP Users. 2007. 119 p.

3. Chang J. Morris. DMMX: Dynamic memory management extensions. Journal of Systems and Software. 2002. 63.3: 192 p.

4. Sapkauskiene Alfreda, Leitoniene Sviesa. The concept of time-based competition in the context of management theory. Engineering Economics. 2010. 212 p.

5. Pulliam Thomas. Time accuracy and the use of implicit methods. In: 11th Computational Fluid Dynamics Conference. 1993. 360 p.

6. Abrossimov E., Rozier Marc, Shapiro Marc. Generic virtual memory management for operating system kernels. In: Proceedings of the twelfth ACM symposium on Operating systems principles. 1989. 136 p.

7. Alonso David Atienza. Dynamic memory management for embedded systems. Springer International Publishing. 2015. 243 p.

8. Comparing Memory Allocation Methods URL: https://learn.microsoft.com/en-gb/windows/win32/memory/comparing-memory-allocation-methods (дата звернення 30.09.2024).

9. Ogihara Mitsunori. Classes String and StringBuilder. Fundamentals of Java Programming. 2018. 234 p.

10. Vömel Stefan, Freiling Felix C. A survey of main memory acquisition and analysis techniques for the windows operating system. Digital Investigation. 2011. 18 p.

11. Kamp Hans. The paradox of the heap. In: Meaning and the Dynamics of Interpretation. Brill. 2013. 311 p.

12. VirtualAlloc function URL: https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc (дата звернення 30.09.2024).

13. VirtualFree function URL: https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualfree (дата звернення 30.09.2024).

14. HeapAlloc function URL: https://learn.microsoft.com/en-us/windows/win32/api/heapapi/nf-heapapi-heapalloc (дата звернення 30.09.2024).

15. HeapFree function URL: https://learn.microsoft.com/en-us/windows/win32/api/heapapi/nf-heapapi-heapfree (дата звернення 30.09.2024).

16. Paging in Operating System. URL: https://www.geeksforgeeks.org/paging-in-operating-system/ (дата звернення 30.09.2024).

17. Paged and Non-paged Pool URL: http://blogmssql.blogspot.com/2014/12/paged-and-non-paged-pool.html (дата звернення 30.09.2024).

18. Valgrind. Software Testing Tools Guide. URL: http://www.testingtoolsguide.net/tools/valgrind/ (дата звернення 30.09.2024).

19. Pros and Cons of Valgrind 2024 URL: https://www.trustradius.com/products/valgrind/reviews?qs=pros-and-cons (дата звернення 30.09.2024).

20. HeapMemView. Description. URL: https://www.nirsoft.net/utils/heap_memory_view.html (дата звернення 30.09.2024).

21. Windows Performance Monitor Overview. URL: https://techcommunity.microsoft.com/t5/ask-the-performance-team/windows-performance-monitor-overview/ba-p/375481 (дата звернення 30.09.2024).

22. What Is The Importance Of Performance Monitoring & Best Tools URL: https://www.timechamp.io/blogs/what-is-the-importance-of-performance-monitoring-best-tools/ (дата звернення 30.09.2024).

23. Ganesh, R., & Prabu, G. Determination of internet banking usage and purpose with explanation of data flow diagram and use case diagram. International Journal of Management and Humanities, 4(7), 2020. pp. 52-58.

24. Iqbal, S., Al-Azzoni, I., Allen, G., & Khan, H. U. Extending UML use case diagrams to represent non-interactive functional requirements. E-Informatica Software Engineering Journal. 2020. Vol. 14 №1. pp. 97-115.

25. Mastrodomenico. R. The python book. John Wiley & Sons. 2022. 1531 p.

26. Гальперін А., Кочнев Д. Робота з Eclipse: Навчальний посібник. - Харків: Видавництво Харківського національного університету імені В.Н. Каразіна, 2017. - 230 с.

27. Троелсен Е. C# 7.0 та платформа .NET: Посібник для професіоналів. - Львів: Видавництво Львівської політехніки, 2018. - 918 с.

28. Price M. J. C# 11 and . NET 7 - Modern Cross-Platform Development Fundamentals: Start Building Websites and Services with ASP. NET Core 7, Blazor, and EF Core 7, 7th Edition. Packt Publishing, Limited, 2022. 826 p.

29. Svyatkovskiy, A., Deng, S. K., Fu, S., & Sundaresan, N. Intellicode compose: Code generation using transformer. In Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering 2020. pp. 1433-1443.

30. Kozma, M., Vincúr, J., & Kapec, P. CollaVRation: An Immersive Virtual Environment for Collaborative Software Development. In Science and Information Conference. Cham: Springer Nature Switzerland. 2024. pp. 280-298.

Виправити поля по всій роботі

Оформити додатки згідно вимог

ДОДАТКИ

~~Додаток А –~~ Технічне завдання

ЗАТВЕРДЖЕНО
1116130.01432-01-ЛЗ

ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ РОБОТИ З ДИНАМІЧНОЮ ПАМ'ЯТТЮ В ОС WINDOWS

Технічне завдання

1116130.01432-01

Листів 7

2025

1. ВВЕДЕННЯ

Програмний комплекс для дослідження часової ефективності роботи з динамічною пам’яттю в ОС Windows призначений для аналізу механізмів виділення, управління та звільнення пам’яті, що використовуються операційною системою. Продукт забезпечує моделювання та тестування різних сценаріїв роботи з динамічною пам’яттю, а також збір і аналіз даних про швидкодію та ефективність цих процесів.

Причиною створення продукту є необхідність удосконалення процесів оптимізації роботи з динамічною пам’яттю, що мають вирішальне значення для продуктивності програмного забезпечення, а також обмежена кількість доступних досліджень у цій сфері.

Область застосування – інженери програмного забезпечення, системні програмісти та дослідники, які займаються оптимізацією роботи операційних систем і додатків.

2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є наказ від 07.12.22 №1209ст ректора Українського державного університету науки і технологій “Про призначення наукових керівників та затвердження тем магістерських робіт” за спеціальністю 121 “Інженерія програмного забезпечення” факультету “Комп’ютерних технологій і систем” по кафедрі “Комп’ютерні інформаційні технології”.

Тема дипломної роботи – “Дослідження часової ефективності роботи з динамічною пам'яттю в ОС Windows”. Керівник – Андрющенко В. О.

3. ПРИЗНАЧЕННЯ РОЗРОБКИ

Функціональне призначення: Продукт, що розробляється, дозволяє інженерам та дослідникам оцінювати часову ефективність роботи з динамічною пам'яттю в ОС Windows. Він забезпечує можливість моделювання різних сценаріїв використання пам’яті, аналізу швидкодії механізмів виділення та звільнення пам’яті, а також виявлення вузьких місць у процесах управління пам’яттю.

Експлуатаційне призначення: Ефективне управління динамічною пам'яттю є важливим фактором для розробки продуктивного програмного забезпечення. Однак, інженери програмного забезпечення часто стикаються з проблемою недостатнього розуміння механізмів роботи пам'яті в ОС Windows. Продукт допомагає проводити дослідження часових характеристик пам'яті в реальних умовах експлуатації, що сприятиме оптимізації програм і підвищенню їхньої продуктивності.

4. ВИМОГИ ДО ПРОГРАМИ

4.1. Вимоги до функціональних характеристик

Можливість моделювати різні сценарії роботи з динамічною пам’яттю; можливість вибору параметрів тестування, таких як розмір виділення, частота запитів пам’яті, кількість потоків; можливість аналізувати часові характеристики виділення та звільнення пам’яті; генерація звітів із результатами тестування.

4.2. Вимоги до надійності

Забезпечення стійкого функціонування програми при виконанні багатопотокових тестів; контроль коректності введених параметрів і цілісності даних; наявність механізму резервного копіювання важливих результатів тестування на зовнішній носій.

4.3. Умови експлуатації

Вимоги до кліматичних умов: температура – 21-25 °С, відносна вологість 40-60%. Обслуговування не потрібне. Для роботи із ПЗ достатньо однієї людини, яка має базовий досвід роботи з ПК і цікавиться дослідженнями у сфері ефективності управління пам’яттю.

4.4. Вимоги до складу і параметрів технічних засобів

Склад технічних засобів: процесор із тактовою частотою 2 ГГц або вище; 10 Мб місця на накопичувачі; 4 Гб оперативної пам’яті.

4.5. Вимоги до інформаційної і програмної сумісності

Програма має функціонувати під управлінням ОС Windows 10/11. Програма написана мовою програмування C#. Ця мова об’єктно-орієнтована, що дозволяє структурувати код на логічні частини, забезпечує високий рівень безпеки і зручність розробки для ОС Windows. На системі має бути встановлений .NET Framework 4.5 або вище.

4.6. Вимоги до маркування і упаковки

Упаковка програмного продукту, включаючи документацію, повинна бути захищена від пошкоджень (механічних, кліматичних). На упаковці повинна бути зазначена назва продукту, мінімальні системні вимоги. На зворотній стороні упаковки має бути вказаний розробник та його юридична адреса.

4.7. Вимоги до транспортування і зберігання

Транспортування повинно проводитися в захищеній упаковці. Умови зберігання повинні забезпечувати безпеку програмного продукту від пошкоджень і впливу зовнішнього середовища.

5. СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів	Примітка
1	Вступ	15.05.2024	
2	Аналіз сучасного стану дослідження проблеми за науковими літературними джерелами	01.06.2024	
3	Аналіз сучасного стану програмно-апаратного забезпечення, яке потребує вдосконалення для вирішення проблем дослідження	01.07.2024	
4	Постановка задачі, технічне завдання	01.08.2024	30%
5	Техніко-економічні показники	01.09.2024	
6	Розробка інструментальних засобів дослідження	12.09.2024	
7	Виконання досліджень	01.10.2024	60%
8	Оформлення тез доповідей	15.10.2024	
9	Оформлення статті у фаховий журнал	01.11.2024	
10	Оформлення пояснювальної записки	15.11.2024	
11	Розробка демонстраційних матеріалів	18.12.2024	100%
12	Подання кваліфікаційної роботи до кафедри	20.12.2024	
13	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії	23.01.2024	

6. ПОРЯДОК КОНТРОЛЮ І ПРИЙМАННЯ

Контроль виконання здійснює керівник розробки Андрющенко В. О. Прийом здійснюється уповноваженою комісією.

У додатках теж є нумерація сторінок: виправити

Номер сторінки у додатках: по центру

7. БІБЛІОГРАФІЧНИЙ СПИСОК

1. XU Luna. Memtune: Dynamic memory management for in-memory data analytic platforms. In: 2016 IEEE international parallel and distributed processing symposium (IPDPS). IEEE, 2016. 386 p.

2. Jones Richard E. Dynamic memory management: Challenges for today and tomorrow. In: International Lisp Conference. Association of LISP Users. 2007. 119 p.

3. Chang J. Morris. DMMX: Dynamic memory management extensions. Journal of Systems and Software. 2002. 63.3: 192 p.

4. Sapkauskiene Alfreda, Leitoniene Sviesa. The concept of time-based competition in the context of management theory. Engineering Economics. 2010. 212 p.

5. Pulliam Thomas. Time accuracy and the use of implicit methods. In: 11th Computational Fluid Dynamics Conference. 1993. 360 p.

6. Abrossimov E., Rozier Marc, Shapiro Marc. Generic virtual memory management for operating system kernels. In: Proceedings of the twelfth ACM symposium on Operating systems principles. 1989. 136 p.

Додаток В – Текст програми

ЗАТВЕРДЖЕНО
1116130.01432-01-ЛЗ

ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ РОБОТИ З ДИНАМІЧНОЮ ПАМ'ЯТТЮ В ОС WINDOWS

Текст програми

1116130.01432-01

Листів 42

2025

Лістинг 1. Код класу «AllocationAnitializArraysFoms»

```
using DinMemWinApp.AppCode;
using DinMemWinApp.Classes;
using LiveCharts.Wpf;
using LiveCharts;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;
using static System.Windows.Forms.VisualStyles.VisualStyleElement;

namespace DinMemWinApp.Forms.Operations {
  public partial class AllocationAnitializArraysFoms : Form {
    private List<LargeArray> _AllLargeArray = new List<LargeArray>();
    private ValidationMy _Validation = new ValidationMy();

    public AllocationAnitializArraysFoms() {
      InitializeComponent();
      TestPBar.Minimum = 0; // Мінімальне значення прогрес бару
    }

    private void AddNewElementBtn_Click(object sender, EventArgs e) {
      if (IsDataEnteringCorrecting()) {
        LargeArray oneLargeArray = new LargeArray();
        oneLargeArray.NumArrays = Convert.ToInt32(NumArraysTBox.Text);
        oneLargeArray.ArraySize = Convert.ToInt32(ArraySizeTBox.Text);
        _AllLargeArray.Add(oneLargeArray);
        LoadDataInLargeArrayGW(_AllLargeArray);
        CliarData();
      }
    }

    private void TestBtn_Click(object sender, EventArgs e) {
      if (_AllLargeArray.Count >= 2) {
        // Очищаємо логи перед початком нових експериментів
        LogsTBox.Clear();
        // Ініціалізуємо графік перед проведенням експериментів
        InitializeChart();
        // Проведення серії експериментів
        PerformExperiments(_AllLargeArray);
        // Оновлюємо дані графіка після завершення експериментів
        UpdateChartWithData();
      } else {
```

```
    MessageBox.Show(NamesMy.Messages.InsufficientData, "Увага");
  }
}


private void LargeArrayGridView_CellClick(object sender, DataGridViewCellEventArgs e) {
  if (e.ColumnIndex == 3 && LargeArrayGridView[0, e.RowIndex].Value.ToString()
    != _AllLargeArray[0].Message) {
    int num = Convert.ToInt32(LargeArrayGridView[0, e.RowIndex].Value.ToString());
    for (int i = 0; i < _AllLargeArray.Count; i++) {
      if (num == _AllLargeArray[i].Number) {
        _AllLargeArray.RemoveAt(i);
        break;
      }
    }
    LoadDataInLargeArrayGW(_AllLargeArray);
  }
}


// Метод для ініціалізації графіка
private void InitializeChart() {
  // Очищуємо серії та осі
  GraphicsCC.Series.Clear();
  GraphicsCC.AxisX.Clear();
  GraphicsCC.AxisY.Clear();
  // Створюємо серії для графіка
  var executionTimeSeries = new LineSeries {
    Title = "Час виконання",
    Values = new ChartValues<double>()
  };
  var memoryUsedSeries = new LineSeries {
    Title = "Використана пам'ять (MB)",
    Values = new ChartValues<double>()
  };
  // Додаємо серії на графік
  GraphicsCC.Series.Add(executionTimeSeries);
  GraphicsCC.Series.Add(memoryUsedSeries);
  // Додаємо підписи для осі X
  GraphicsCC.AxisX.Add(new LiveCharts.Wpf.Axis {
    Title = "Кількість масивів",
    Labels = _AllLargeArray.Select(ex => ex.NumArrays.ToString()).ToArray(),
    Separator = new LiveCharts.Wpf.Separator { Step = 1 }
  });
  // Вісь Y
  GraphicsCC.AxisY.Add(new LiveCharts.Wpf.Axis {
    Title = "Час виконання (мс)"
  });
}

// Метод для оновлення даних на графіку
private void UpdateChartWithData() {
```

```
      // Переконуємося, що є серії для оновлення
      if (GraphicsCC.Series.Count > 0) {
        var executionTimeSeries = (LineSeries)GraphicsCC.Series[0];
        var memoryUsedSeries = (LineSeries)GraphicsCC.Series[1];
        // Очищуємо старі дані
        executionTimeSeries.Values.Clear();
        memoryUsedSeries.Values.Clear();
        // Додаємо нові дані
        foreach (var experiment in _AllLargeArray) {
          executionTimeSeries.Values.Add(experiment.ExecutionTimeMilliseconds);
          memoryUsedSeries.Values.Add(experiment.MemoryUsedMB);
        }
      }
    }

    private void CliarData() {
      NumArraysTBox.Text = "0";
      ArraySizeTBox.Text = "0";
    }

    // Метод для вимірювання операцій і збереження результатів у LargeArray
    private void MeasureLargeArrayAllocation(LargeArray experiment) {
      PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time",
"_Total");
      PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
      Stopwatch stopwatch = new Stopwatch();
      stopwatch.Start();
      long initialMemory = GC.GetTotalMemory(true);
      double initialCpuUsage = cpuCounter.NextValue();
      for (int i = 0; i < experiment.NumArrays; i++) {
        int[] array = new int[experiment.ArraySize];
        for (int j = 0; j < array.Length; j++) {
          array[j] = j; // Ініціалізація масиву
        }
      }
      stopwatch.Stop();
      long finalMemory = GC.GetTotalMemory(false);
      double finalCpuUsage = cpuCounter.NextValue();
      double availableMemoryMB = ramCounter.NextValue();
      experiment.ExecutionTimeMilliseconds = stopwatch.Elapsed.TotalMilliseconds;
      experiment.MemoryUsedMB = (finalMemory - initialMemory) / (1024 * 1024);
      experiment.AverageCpuUsagePercent = (initialCpuUsage + finalCpuUsage) / 2;
      experiment.AvailableMemoryMB = availableMemoryMB;
    }

    // Метод для відображення результатів у LogsTBox
    private void DisplayResults(ExperimentResult result) {
      // Виведення результатів у LogsTBox замість Console
      LogsTBox.AppendText($"Час виконання: {result.ExecutionTimeMilliseconds}
мс{Environment.NewLine}");
      LogsTBox.AppendText($"Використано пам'яті: {result.MemoryUsedMB}
MB{Environment.NewLine}");
```

```
    LogsTBox.AppendText($"Середнє навантаження процесора:
{result.AverageCpuUsagePercent}%{Environment.NewLine}");
    LogsTBox.AppendText($"Доступна пам'ять: {result.AvailableMemoryMB}
MB{Environment.NewLine}");
    Application.DoEvents();
   }

   // Метод для виконання серії експериментів з оновленням прогрес бару
   private void PerformExperiments(List<LargeArray> experiments) {
    TestPBar.Maximum = experiments.Count; // Встановлюємо максимальне значення для прогрес-
бару
    TestPBar.Value = 0; // Скидаємо прогрес на початок
    foreach (var experiment in experiments) {
     // Виведення номера експерименту та повідомлення перед запуском
     LogsTBox.AppendText($"Експеримент {experiment.Number}:
{experiment.Message}{Environment.NewLine}");
     // Проведення заміру і збереження результатів
     MeasureLargeArrayAllocation(experiment);
     // Виведення результатів після завершення кожного експерименту
     DisplayResults(experiment);
     // Оновлюємо прогрес бар
     TestPBar.Value += 1;
     // Оновлюємо інтерфейс, щоб прогрес бар відображався під час виконання експериментів
     Application.DoEvents();
    }
   }

   private void LoadDataInLargeArrayGW(List<LargeArray> AllLargeArray) {
    ChangeNumber();
    LargeArrayGridView.DataSource = null;
    LargeArrayGridView.Columns.Clear();
    LargeArrayGridView.AutoGenerateColumns = false;
    LargeArrayGridView.RowHeadersVisible = false;

    LargeArrayGridView.DataSource = AllLargeArray;

    if (AllLargeArray.Count > 0) {
     DataGridViewColumn numberColumn = new DataGridViewTextBoxColumn();
     numberColumn.HeaderText = "№ з/п";
     numberColumn.DataPropertyName = "Number";
     numberColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
     numberColumn.Width = NamesMy.SizeOptins.NumberSize;
     LargeArrayGridView.Columns.Add(numberColumn);

     DataGridViewColumn NumArraysColumn = new DataGridViewTextBoxColumn();
     NumArraysColumn.HeaderText = "К-сть масивів";
     NumArraysColumn.DataPropertyName = "NumArrays";
     NumArraysColumn.Name = "NumArrays";
     NumArraysColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
     NumArraysColumn.Width = 120;
     LargeArrayGridView.Columns.Add(NumArraysColumn);
```

```
    DataGridViewColumn ArraySizeColumn = new DataGridViewTextBoxColumn();
    ArraySizeColumn.HeaderText = "Розмір масиву";
    ArraySizeColumn.DataPropertyName = "ArraySize";
    ArraySizeColumn.Name = "ArraySize";
    ArraySizeColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
    ArraySizeColumn.Width = 120;
    LargeArrayGridView.Columns.Add(ArraySizeColumn);

    DataGridViewButtonColumn DeleteBtn = new DataGridViewButtonColumn();
    DeleteBtn.Text = "Видалити";
    DeleteBtn.UseColumnTextForButtonValue = true;
    DeleteBtn.ToolTipText = "Видалити";
    DeleteBtn.Width = NamesMy.SizeOptins.DeleteBtnSize;
    LargeArrayGridView.Columns.Add(DeleteBtn);

    for (int i = 0; i < LargeArrayGridView.Columns.Count; i++) {
      LargeArrayGridView.Columns[i].HeaderCell.Style.BackColor = Color.LightGray;
    }
  }
}

private void ChangeNumber() {
  for (int i = 0; i < _AllLargeArray.Count; i++) {
    _AllLargeArray[i].Number = i + 1;
  }
}

private bool IsDataEnteringCorrecting() {
  bool isCorrect = true;
  if (_Validation.IsDataConvertToInt(NumArraysTBox.Text)) {
    if (_Validation.IsDataInThisScope(1, 10000000, Convert.ToInt32(NumArraysTBox.Text))) {
      NumArraysValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
    } else {
      NumArraysValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
      isCorrect = false;
    }
  } else {
    NumArraysValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
    isCorrect = false;
  }
  if (_Validation.IsDataConvertToInt(ArraySizeTBox.Text)) {
    if (_Validation.IsDataInThisScope(1, 10000000, Convert.ToInt32(ArraySizeTBox.Text))) {
      ArraySizeValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
    } else {
      ArraySizeValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
      isCorrect = false;
    }
  } else {
    ArraySizeValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
    isCorrect = false;
  }
  return isCorrect;
```

```
    }

  }
}


// Клас для збереження результатів експерименту
public class ExperimentResult {
  public double ExecutionTimeMilliseconds { get; set; }
  public double MemoryUsedMB { get; set; }
  public double AverageCpuUsagePercent { get; set; }
  public double AvailableMemoryMB { get; set; }
}
```

Лістинг 2. Код класу «DangerousCodeForm»

```
using DinMemWinApp.AppCode;
using DinMemWinApp.Classes;
using LiveCharts.Wpf;
using LiveCharts;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace DinMemWinApp.Forms.Operations {
  public partial class DangerousCodeForm : Form {
    private List<DangerousCode> _AllDangerousCode = new List<DangerousCode>();
    private ValidationMy _Validation = new ValidationMy();

    public DangerousCodeForm() {
      InitializeComponent();
      TestPBar.Minimum = 0; // Мінімальне значення прогрес бару
    }

    private void AddNewElementBtn_Click(object sender, EventArgs e) {
      if (IsDataEnteringCorrecting()) {
        DangerousCode oneDangerousCode = new DangerousCode();
        oneDangerousCode.OperationCount = Convert.ToInt32(OperationCountTBox.Text);
        _AllDangerousCode.Add(oneDangerousCode);
        LoadDataInLargeArrayGW(_AllDangerousCode);
        CliarData();
      }
    }

    private void TestBtn_Click(object sender, EventArgs e) {
      if (_AllDangerousCode.Count >= 2) {
        // Очищаємо логи перед початком нових експериментів
```

```
        LogsTBox.Clear();

        // Ініціалізуємо графік перед проведенням експериментів
        InitializeChart();

        // Проведення серії експериментів
        PerformExperimentsWithDangerousCode(_AllDangerousCode);

        // Оновлюємо дані графіка після завершення експериментів
        UpdateChartWithData();
      } else {
        MessageBox.Show(NamesMy.Messages.InsufficientData, "Увага");
      }
    }

    private void DangerousCodeGridView_CellClick(object sender, DataGridViewCellEventArgs e) {
      if (e.ColumnIndex == 2 && DangerousCodeGridView[0, e.RowIndex].Value.ToString() !=
_AllDangerousCode[0].Message) {
        int num = Convert.ToInt32(DangerousCodeGridView[0, e.RowIndex].Value.ToString());
        for (int i = 0; i < _AllDangerousCode.Count; i++) {
          if (num == _AllDangerousCode[i].Number) {
            _AllDangerousCode.RemoveAt(i);
            break;
          }
        }
        LoadDataInLargeArrayGW(_AllDangerousCode);
      }
    }

    private void CliarData() {
      OperationCountTBox.Text = "0";
    }

    // Метод для ініціалізації графіка
    private void InitializeChart() {
      // Очищуємо серії та осі
      GraphicsCC.Series.Clear();
      GraphicsCC.AxisX.Clear();
      GraphicsCC.AxisY.Clear();

      // Створюємо серії для графіка
      var executionTimeSeries = new LineSeries {
        Title = "Час виконання",
        Values = new ChartValues<double>()
      };

      var memoryUsedSeries = new LineSeries {
        Title = "Використана пам'ять (MB)",
        Values = new ChartValues<double>()
      };

      // Додаємо серії на графік
```

```
    GraphicsCC.Series.Add(executionTimeSeries);
    GraphicsCC.Series.Add(memoryUsedSeries);

    // Додаємо підписи для осі X
    GraphicsCC.AxisX.Add(new LiveCharts.Wpf.Axis {
      Title = "Кількість операцій",
      Labels = _AllDangerousCode.Select(ex => ex.OperationCount.ToString()).ToArray(),
      Separator = new LiveCharts.Wpf.Separator { Step = 1 }
    });

    // Вісь Y
    GraphicsCC.AxisY.Add(new LiveCharts.Wpf.Axis {
      Title = "Час виконання (мс)"
    });
  }

  // Метод для оновлення даних на графіку
  private void UpdateChartWithData() {
    // Переконуємося, що є серії для оновлення
    if (GraphicsCC.Series.Count > 0) {
      var executionTimeSeries = (LineSeries)GraphicsCC.Series[0];
      var memoryUsedSeries = (LineSeries)GraphicsCC.Series[1];

      // Очищуємо старі дані
      executionTimeSeries.Values.Clear();
      memoryUsedSeries.Values.Clear();

      // Додаємо нові дані
      foreach (var experiment in _AllDangerousCode) {
        executionTimeSeries.Values.Add(experiment.ExecutionTimeMilliseconds);
        memoryUsedSeries.Values.Add(experiment.MemoryUsedMB);
      }
    }
  }

  // Метод для вимірювання операцій з використанням небезпечного коду
  private unsafe void MeasureDangerousCodeAllocation(DangerousCode experiment) {
    PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time",
"_Total");
    PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
    Stopwatch stopwatch = new Stopwatch();
    long initialMemory = GC.GetTotalMemory(true);
    double initialCpuUsage = cpuCounter.NextValue();
    // Виділення пам'яті на масив
    int[] array = new int[experiment.OperationCount];
    // Ініціалізація масиву
    for (int i = 0; i < array.Length; i++) {
      array[i] = i;
    }
    // Доступ за допомогою небезпечного коду (покажчики)
    unsafe {
      fixed (int* ptr = array) {
```

```
        stopwatch.Start();
        for (int i = 0; i < experiment.OperationCount; i++) {
          // Доступ до елементів через покажчик
          int value = *(ptr + i);
        }
        stopwatch.Stop();
      }
    }
    long finalMemory = GC.GetTotalMemory(false);
    double finalCpuUsage = cpuCounter.NextValue();
    double availableMemoryMB = ramCounter.NextValue();
    // Збереження результатів
    experiment.ExecutionTimeMilliseconds = stopwatch.Elapsed.TotalMilliseconds;
    experiment.MemoryUsedMB = (finalMemory - initialMemory) / (1024 * 1024);
    experiment.AverageCpuUsagePercent = (initialCpuUsage + finalCpuUsage) / 2;
    experiment.AvailableMemoryMB = availableMemoryMB;
  }



  // Метод для відображення результатів
  private void DisplayResults(ExperimentResult result) {
    // Виведення результатів у LogsTBox або Console
    LogsTBox.AppendText($"Час виконання: {result.ExecutionTimeMilliseconds}
мс{Environment.NewLine}");
    LogsTBox.AppendText($"Використано пам'яті: {result.MemoryUsedMB}
MB{Environment.NewLine}");
    LogsTBox.AppendText($"Середнє навантаження процесора:
{result.AverageCpuUsagePercent}%{Environment.NewLine}");
    LogsTBox.AppendText($"Доступна пам'ять: {result.AvailableMemoryMB}
MB{Environment.NewLine}");
    Application.DoEvents(); // Оновлюємо інтерфейс
  }

  // Метод для виконання серії експериментів
  private void PerformExperimentsWithDangerousCode(List<DangerousCode> experiments) {
    TestPBar.Maximum = experiments.Count; // Встановлюємо максимальне значення для прогрес
бару
    TestPBar.Value = 0; // Скидаємо прогрес на початок

    foreach (var experiment in experiments) {
      // Виведення номера експерименту та повідомлення перед запуском
      LogsTBox.AppendText($"Експеримент {experiment.Number}:
{experiment.Message}{Environment.NewLine}");

      // Проведення заміру і збереження результатів
      MeasureDangerousCodeAllocation(experiment);

      // Виведення результатів після завершення кожного експерименту
      DisplayResults(experiment);

      // Оновлення прогрес бару
```

```
        TestPBar.Value += 1;
        Application.DoEvents(); // Оновлення інтерфейсу
      }
    }


    private void LoadDataInLargeArrayGW(List<DangerousCode> AllDangerousCode) {
      ChangeNumber();
      DangerousCodeGridView.DataSource = null;
      DangerousCodeGridView.Columns.Clear();
      DangerousCodeGridView.AutoGenerateColumns = false;
      DangerousCodeGridView.RowHeadersVisible = false;

      DangerousCodeGridView.DataSource = AllDangerousCode;

      if (AllDangerousCode.Count > 0) {
        DataGridViewColumn numberColumn = new DataGridViewTextBoxColumn();
        numberColumn.HeaderText = "№ з/п";
        numberColumn.DataPropertyName = "Number";
        numberColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
        numberColumn.Width = NamesMy.SizeOptins.NumberSize;
        DangerousCodeGridView.Columns.Add(numberColumn);

        DataGridViewColumn OperationCountColumn = new DataGridViewTextBoxColumn();
        OperationCountColumn.HeaderText = "К-сть операцій";
        OperationCountColumn.DataPropertyName = "OperationCount";
        OperationCountColumn.Name = "OperationCount";
        OperationCountColumn.DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleRight;
        OperationCountColumn.Width = 120;
        DangerousCodeGridView.Columns.Add(OperationCountColumn);

        DataGridViewButtonColumn DeleteBtn = new DataGridViewButtonColumn();
        DeleteBtn.Text = "Видалити";
        DeleteBtn.UseColumnTextForButtonValue = true;
        DeleteBtn.ToolTipText = "Видалити";
        DeleteBtn.Width = NamesMy.SizeOptins.DeleteBtnSize;
        DangerousCodeGridView.Columns.Add(DeleteBtn);

        for (int i = 0; i < DangerousCodeGridView.Columns.Count; i++) {
          DangerousCodeGridView.Columns[i].HeaderCell.Style.BackColor = Color.LightGray;
        }
      }
    }

    private void ChangeNumber() {
      for (int i = 0; i < _AllDangerousCode.Count; i++) {
        _AllDangerousCode[i].Number = i + 1;
      }
    }

    private bool IsDataEnteringCorrecting() {
```

```
    bool isCorrect = true;
    if (_Validation.IsDataConvertToInt(OperationCountTBox.Text)) {
      if (_Validation.IsDataInThisScope(1, 10000000, Convert.ToInt32(OperationCountTBox.Text))) {
        DynamicStructValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
      } else {
        DynamicStructValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
        isCorrect = false;
      }
    } else {
      DynamicStructValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
      isCorrect = false;
    }
    return isCorrect;
  }



 }
}
```

Лістинг 3. Код класу «DynamicListForm»

```
using DinMemWinApp.AppCode;
using DinMemWinApp.Classes;
using LiveCharts.Wpf;
using LiveCharts;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Diagnostics;

namespace DinMemWinApp.Forms.Operations {
 public partial class DynamicListForm : Form {
  private List<DynamicStruct> _AllDynamicStruct = new List<DynamicStruct>();
  private ValidationMy _Validation = new ValidationMy();

  public DynamicListForm() {
   InitializeComponent();
   TestPBar.Minimum = 0; // Мінімальне значення прогрес бару
  }

  private void AddNewElementBtn_Click(object sender, EventArgs e) {
   if (IsDataEnteringCorrecting()) {
    DynamicStruct oneDynamicStruct = new DynamicStruct();
    oneDynamicStruct.OperationCount = Convert.ToInt32(OperationCountTBox.Text);
    _AllDynamicStruct.Add(oneDynamicStruct);
    LoadDataInLargeArrayGW(_AllDynamicStruct);
```

```
      CliarData();
    }
  }

  private void TestBtn_Click(object sender, EventArgs e) {
    if (_AllDynamicStruct.Count >= 2) {
      // Очищаємо логи перед початком нових експериментів
      LogsTBox.Clear();

      // Ініціалізуємо графік перед проведенням експериментів
      InitializeChart();

      // Проведення серії експериментів
      PerformExperimentsWithDynamicStruct(_AllDynamicStruct);

      // Оновлюємо дані графіка після завершення експериментів
      UpdateChartWithData();
    } else {
      MessageBox.Show(NamesMy.Messages.InsufficientData, "Увага");
    }
  }

  private void DynamicStructGridView_CellClick(object sender, DataGridViewCellEventArgs e) {
    if (e.ColumnIndex == 2 && DynamicStructGridView[0, e.RowIndex].Value.ToString() !=
_AllDynamicStruct[0].Message) {
      int num = Convert.ToInt32(DynamicStructGridView[0, e.RowIndex].Value.ToString());
      for (int i = 0; i < _AllDynamicStruct.Count; i++) {
        if (num == _AllDynamicStruct[i].Number) {
          _AllDynamicStruct.RemoveAt(i);
          break;
        }
      }
      LoadDataInLargeArrayGW(_AllDynamicStruct);
    }
  }

  private void CliarData() {
    OperationCountTBox.Text = "0";
  }

  // Метод для ініціалізації графіка
  private void InitializeChart() {
    // Очищуємо серії та осі
    GraphicsCC.Series.Clear();
    GraphicsCC.AxisX.Clear();
    GraphicsCC.AxisY.Clear();

    // Створюємо серії для графіка
    var executionTimeSeries = new LineSeries {
      Title = "Час виконання",
      Values = new ChartValues<double>()
    };
```

```
    var memoryUsedSeries = new LineSeries {
      Title = "Використана пам'ять (MB)",
      Values = new ChartValues<double>()
    };

    // Додаємо серії на графік
    GraphicsCC.Series.Add(executionTimeSeries);
    GraphicsCC.Series.Add(memoryUsedSeries);

    // Додаємо підписи для осі X
    GraphicsCC.AxisX.Add(new LiveCharts.Wpf.Axis {
      Title = "Кількість елементів",
      Labels = _AllDynamicStruct.Select(ex => ex.OperationCount.ToString()).ToArray(),
      Separator = new LiveCharts.Wpf.Separator { Step = 1 }
    });

    // Вісь Y
    GraphicsCC.AxisY.Add(new LiveCharts.Wpf.Axis {
      Title = "Час виконання (мс)"
    });
  }

  // Метод для оновлення даних на графіку
  private void UpdateChartWithData() {
    // Переконуємося, що є серії для оновлення
    if (GraphicsCC.Series.Count > 0) {
      var executionTimeSeries = (LineSeries)GraphicsCC.Series[0];
      var memoryUsedSeries = (LineSeries)GraphicsCC.Series[1];

      // Очищуємо старі дані
      executionTimeSeries.Values.Clear();
      memoryUsedSeries.Values.Clear();

      // Додаємо нові дані
      foreach (var experiment in _AllDynamicStruct) {
        executionTimeSeries.Values.Add(experiment.ExecutionTimeMilliseconds);
        memoryUsedSeries.Values.Add(experiment.MemoryUsedMB);
      }
    }
  }

  // Метод для вимірювання операцій і збереження результатів у DynamicStruct
  private void MeasureDynamicStructAllocation(DynamicStruct experiment) {
    PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time",
"_Total");
    PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    long initialMemory = GC.GetTotalMemory(true);
    double initialCpuUsage = cpuCounter.NextValue();
    // Операції з динамічним списком
```

```
    List<int> dynamicList = new List<int>();
    for (int i = 0; i < experiment.OperationCount; i++) {
      // Додаємо елементи до динамічного списку
      dynamicList.Add(i);
    }
    // Перевірка операцій над списком: видалення, доступ до елементів, очищення
    if (dynamicList.Count > 0) {
      // Доступ до першого елемента
      int firstElement = dynamicList[0];
      // Видалення останнього елемента
      dynamicList.RemoveAt(dynamicList.Count - 1);
      // Очищення списку
      dynamicList.Clear();
    }
    stopwatch.Stop();
    long finalMemory = GC.GetTotalMemory(false);
    double finalCpuUsage = cpuCounter.NextValue();
    double availableMemoryMB = ramCounter.NextValue();
    experiment.ExecutionTimeMilliseconds = stopwatch.Elapsed.TotalMilliseconds;
    experiment.MemoryUsedMB = (finalMemory - initialMemory) / (1024 * 1024);
    experiment.AverageCpuUsagePercent = (initialCpuUsage + finalCpuUsage) / 2;
    experiment.AvailableMemoryMB = availableMemoryMB;
  }


  // Метод для відображення результатів у текстовому боксі або в консолі
  private void DisplayResults(ExperimentResult result) {
    // Виведення результатів у LogsTBox або Console
    LogsTBox.AppendText($"Час виконання: {result.ExecutionTimeMilliseconds}
мс{Environment.NewLine}");
    LogsTBox.AppendText($"Використано пам'яті: {result.MemoryUsedMB}
MB{Environment.NewLine}");
    LogsTBox.AppendText($"Середнє навантаження процесора:
{result.AverageCpuUsagePercent}%{Environment.NewLine}");
    LogsTBox.AppendText($"Доступна пам'ять: {result.AvailableMemoryMB}
MB{Environment.NewLine}");
    Application.DoEvents();
  }

  // Метод для виконання серії експериментів з оновленням прогрес бару
  private void PerformExperimentsWithDynamicStruct(List<DynamicStruct> experiments) {
    TestPBar.Maximum = experiments.Count; // Встановлюємо максимальне значення для прогрес
бару
    TestPBar.Value = 0; // Скидаємо прогрес на початок

    foreach (var experiment in experiments) {
      // Виведення номера експерименту та повідомлення перед запуском
      LogsTBox.AppendText($"Експеримент {experiment.Number}:
{experiment.Message}{Environment.NewLine}");

      // Проведення заміру і збереження результатів
      MeasureDynamicStructAllocation(experiment);
```

```
        // Виведення результатів після завершення кожного експерименту
        DisplayResults(experiment);

        // Оновлення прогрес бару
        TestPBar.Value += 1;
        Application.DoEvents(); // Оновлення інтерфейсу
      }
    }


    private void LoadDataInLargeArrayGW(List<DynamicStruct> AllDynamicStruct) {
      ChangeNumber();
      DynamicStructGridView.DataSource = null;
      DynamicStructGridView.Columns.Clear();
      DynamicStructGridView.AutoGenerateColumns = false;
      DynamicStructGridView.RowHeadersVisible = false;

      DynamicStructGridView.DataSource = AllDynamicStruct;

      if (AllDynamicStruct.Count > 0) {
        DataGridViewColumn numberColumn = new DataGridViewTextBoxColumn();
        numberColumn.HeaderText = "№ з/п";
        numberColumn.DataPropertyName = "Number";
        numberColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
        numberColumn.Width = NamesMy.SizeOptins.NumberSize;
        DynamicStructGridView.Columns.Add(numberColumn);

        DataGridViewColumn OperationCountColumn = new DataGridViewTextBoxColumn();
        OperationCountColumn.HeaderText = "К-сть елементів структури";
        OperationCountColumn.DataPropertyName = "OperationCount";
        OperationCountColumn.Name = "OperationCount";
        OperationCountColumn.DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleRight;
        OperationCountColumn.Width = 120;
        DynamicStructGridView.Columns.Add(OperationCountColumn);

        DataGridViewButtonColumn DeleteBtn = new DataGridViewButtonColumn();
        DeleteBtn.Text = "Видалити";
        DeleteBtn.UseColumnTextForButtonValue = true;
        DeleteBtn.ToolTipText = "Видалити";
        DeleteBtn.Width = NamesMy.SizeOptins.DeleteBtnSize;
        DynamicStructGridView.Columns.Add(DeleteBtn);

        for (int i = 0; i < DynamicStructGridView.Columns.Count; i++) {
          DynamicStructGridView.Columns[i].HeaderCell.Style.BackColor = Color.LightGray;
        }
      }
    }

    private void ChangeNumber() {
      for (int i = 0; i < _AllDynamicStruct.Count; i++) {
```

```
      _AllDynamicStruct[i].Number = i + 1;
    }
  }

  private bool IsDataEnteringCorrecting() {
    bool isCorrect = true;
    if (_Validation.IsDataConvertToInt(OperationCountTBox.Text)) {
      if (_Validation.IsDataInThisScope(1, 10000000, Convert.ToInt32(OperationCountTBox.Text))) {
        DynamicStructValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
      } else {
        DynamicStructValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
        isCorrect = false;
      }
    } else {
      DynamicStructValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
      isCorrect = false;
    }
    return isCorrect;
  }


 }
}
```

Лістинг 4. Код класу «LinkedListsForm»

```
using DinMemWinApp.AppCode;
using DinMemWinApp.Classes;
using LiveCharts.Wpf;
using LiveCharts;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace DinMemWinApp.Forms.Operations {
 public partial class LinkedListsForm : Form {
  private List<LinkedListsOperation> _AllLinkedListsOperation = new List<LinkedListsOperation>();
  private ValidationMy _Validation = new ValidationMy();

  public LinkedListsForm() {
    InitializeComponent();
    TestPBar.Minimum = 0; // Мінімальне значення прогрес бару
  }

  private void AddNewElementBtn_Click(object sender, EventArgs e) {
    if (IsDataEnteringCorrecting()) {
```

```
        LinkedListsOperation oneLinkedListsOperation = new LinkedListsOperation();
        oneLinkedListsOperation.OperationCount = Convert.ToInt32(OperationCountTBox.Text);
        _AllLinkedListsOperation.Add(oneLinkedListsOperation);
        LoadDataInLargeArrayGW(_AllLinkedListsOperation);
        CliarData();
      }
    }

    private void TestBtn_Click(object sender, EventArgs e) {
      if (_AllLinkedListsOperation.Count >= 2) {
        // Очищаємо логи перед початком нових експериментів
        LogsTBox.Clear();

        // Ініціалізуємо графік перед проведенням експериментів
        InitializeChart();

        // Проведення серії експериментів
        PerformExperimentsWithLinkedLists(_AllLinkedListsOperation);

        // Оновлюємо дані графіка після завершення експериментів
        UpdateChartWithData();
      } else {
        MessageBox.Show(NamesMy.Messages.InsufficientData, "Увага");
      }
    }

    private void LinkedListsOperationGridView_CellClick(object sender, DataGridViewCellEventArgs e)
{
      if (e.ColumnIndex == 2 && LinkedListsOperationGridView[0, e.RowIndex].Value.ToString() !=
_AllLinkedListsOperation[0].Message) {
        int num = Convert.ToInt32(LinkedListsOperationGridView[0, e.RowIndex].Value.ToString());
        for (int i = 0; i < _AllLinkedListsOperation.Count; i++) {
          if (num == _AllLinkedListsOperation[i].Number) {
            _AllLinkedListsOperation.RemoveAt(i);
            break;
          }
        }
        LoadDataInLargeArrayGW(_AllLinkedListsOperation);
      }
    }

    private void CliarData() {
      OperationCountTBox.Text = "0";
    }

    // Метод для ініціалізації графіка
    private void InitializeChart() {
      // Очищуємо серії та осі
      GraphicsCC.Series.Clear();
      GraphicsCC.AxisX.Clear();
      GraphicsCC.AxisY.Clear();
```

```
    // Створюємо серії для графіка
    var executionTimeSeries = new LineSeries {
      Title = "Час виконання",
      Values = new ChartValues<double>()
    };

    var memoryUsedSeries = new LineSeries {
      Title = "Використана пам'ять (MB)",
      Values = new ChartValues<double>()
    };

    // Додаємо серії на графік
    GraphicsCC.Series.Add(executionTimeSeries);
    GraphicsCC.Series.Add(memoryUsedSeries);

    // Додаємо підписи для осі X
    GraphicsCC.AxisX.Add(new LiveCharts.Wpf.Axis {
      Title = "Кількість елементів",
      Labels = _AllLinkedListsOperation.Select(ex => ex.OperationCount.ToString()).ToArray(),
      Separator = new LiveCharts.Wpf.Separator { Step = 1 }
    });

    // Вісь Y
    GraphicsCC.AxisY.Add(new LiveCharts.Wpf.Axis {
      Title = "Час виконання (мс)"
    });
  }

  // Метод для оновлення даних на графіку
  private void UpdateChartWithData() {
    // Переконуємося, що є серії для оновлення
    if (GraphicsCC.Series.Count > 0) {
      var executionTimeSeries = (LineSeries)GraphicsCC.Series[0];
      var memoryUsedSeries = (LineSeries)GraphicsCC.Series[1];

      // Очищуємо старі дані
      executionTimeSeries.Values.Clear();
      memoryUsedSeries.Values.Clear();

      // Додаємо нові дані
      foreach (var experiment in _AllLinkedListsOperation) {
        executionTimeSeries.Values.Add(experiment.ExecutionTimeMilliseconds);
        memoryUsedSeries.Values.Add(experiment.MemoryUsedMB);
      }
    }
  }

  private void MeasureLinkedListOperations(LinkedListsOperation experiment) {
    PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time",
"_Total");
    PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
    Stopwatch stopwatch = new Stopwatch();
```

```
    stopwatch.Start();
    long initialMemory = GC.GetTotalMemory(true);
    double initialCpuUsage = cpuCounter.NextValue();
    // Ініціалізація пов'язаного списку
    LinkedList<int> linkedList = new LinkedList<int>();
    // Додавання елементів у пов'язаний список
    for (int i = 0; i < experiment.OperationCount; i++) {
      linkedList.AddLast(i); // Додаємо елемент в кінець списку
    }
    // Операції з пов'язаним списком: доступ до елементів і видалення
    if (linkedList.Count > 0) {
      // Доступ до першого елемента
      int firstElement = linkedList.First.Value;
      // Видалення останнього елемента
      linkedList.RemoveLast();
      // Очищення списку
      linkedList.Clear();
    }
    stopwatch.Stop();
    long finalMemory = GC.GetTotalMemory(false);
    double finalCpuUsage = cpuCounter.NextValue();
    double availableMemoryMB = ramCounter.NextValue();
    // Збереження результатів експерименту
    experiment.ExecutionTimeMilliseconds = stopwatch.Elapsed.TotalMilliseconds;
    experiment.MemoryUsedMB = (finalMemory - initialMemory) / (1024 * 1024);
    experiment.AverageCpuUsagePercent = (initialCpuUsage + finalCpuUsage) / 2;
    experiment.AvailableMemoryMB = availableMemoryMB;
  }



  // Метод для відображення результатів у текстовому боксі або в консолі
  private void DisplayResults(ExperimentResult result) {
    // Виведення результатів у LogsTBox або Console
    LogsTBox.AppendText($"Час виконання: {result.ExecutionTimeMilliseconds}
мс{Environment.NewLine}");
    LogsTBox.AppendText($"Використано пам'яті: {result.MemoryUsedMB}
MB{Environment.NewLine}");
    LogsTBox.AppendText($"Середнє навантаження процесора:
{result.AverageCpuUsagePercent}%{Environment.NewLine}");
    LogsTBox.AppendText($"Доступна пам'ять: {result.AvailableMemoryMB}
MB{Environment.NewLine}");
    Application.DoEvents();
  }

  // Метод для виконання серії експериментів з оновленням прогрес бару
  private void PerformExperimentsWithLinkedLists(List<LinkedListsOperation> experiments) {
    TestPBar.Maximum = experiments.Count; // Встановлюємо максимальне значення для прогрес-
бару
    TestPBar.Value = 0; // Скидаємо прогрес на початок

    foreach (var experiment in experiments) {
```

```
        // Виведення номера експерименту та повідомлення перед запуском
        LogsTBox.AppendText($"Експеримент {experiment.Number}:
{experiment.Message}{Environment.NewLine}");

        // Проведення заміру і збереження результатів
        MeasureLinkedListOperations(experiment);

        // Виведення результатів після завершення кожного експерименту
        DisplayResults(experiment);

        // Оновлення прогрес-бару
        TestPBar.Value += 1;
        Application.DoEvents(); // Оновлення інтерфейсу
      }
    }



    private void LoadDataInLargeArrayGW(List<LinkedListsOperation> AllLinkedListsOperation) {
      ChangeNumber();
      LinkedListsOperationGridView.DataSource = null;
      LinkedListsOperationGridView.Columns.Clear();
      LinkedListsOperationGridView.AutoGenerateColumns = false;
      LinkedListsOperationGridView.RowHeadersVisible = false;

      LinkedListsOperationGridView.DataSource = AllLinkedListsOperation;

      if (AllLinkedListsOperation.Count > 0) {
        DataGridViewColumn numberColumn = new DataGridViewTextBoxColumn();
        numberColumn.HeaderText = "№ з/п";
        numberColumn.DataPropertyName = "Number";
        numberColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
        numberColumn.Width = NamesMy.SizeOptins.NumberSize;
        LinkedListsOperationGridView.Columns.Add(numberColumn);

        DataGridViewColumn OperationCountColumn = new DataGridViewTextBoxColumn();
        OperationCountColumn.HeaderText = "К-сть елементів списку";
        OperationCountColumn.DataPropertyName = "OperationCount";
        OperationCountColumn.Name = "OperationCount";
        OperationCountColumn.DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleRight;
        OperationCountColumn.Width = 120;
        LinkedListsOperationGridView.Columns.Add(OperationCountColumn);

        DataGridViewButtonColumn DeleteBtn = new DataGridViewButtonColumn();
        DeleteBtn.Text = "Видалити";
        DeleteBtn.UseColumnTextForButtonValue = true;
        DeleteBtn.ToolTipText = "Видалити";
        DeleteBtn.Width = NamesMy.SizeOptins.DeleteBtnSize;
        LinkedListsOperationGridView.Columns.Add(DeleteBtn);

        for (int i = 0; i < LinkedListsOperationGridView.Columns.Count; i++) {
```

```
        LinkedListsOperationGridView.Columns[i].HeaderCell.Style.BackColor = Color.LightGray;
       }
      }
    }

    private void ChangeNumber() {
      for (int i = 0; i < _AllLinkedListsOperation.Count; i++) {
        _AllLinkedListsOperation[i].Number = i + 1;
      }
    }

    private bool IsDataEnteringCorrecting() {
      bool isCorrect = true;
      if (_Validation.IsDataConvertToInt(OperationCountTBox.Text)) {
        if (_Validation.IsDataInThisScope(1, 10000000, Convert.ToInt32(OperationCountTBox.Text))) {
          DynamicStructValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
        } else {
          DynamicStructValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
          isCorrect = false;
        }
      } else {
        DynamicStructValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
        isCorrect = false;
      }
      return isCorrect;
    }



  }
}
```

Лістинг 5. Код класу «MemoryFragmentationForm»

```
using DinMemWinApp.AppCode;
using DinMemWinApp.Classes;
using LiveCharts.Wpf;
using LiveCharts;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace DinMemWinApp.Forms.Operations {
  public partial class MemoryFragmentationForm : Form {
    private List<MemoryFragmentation> _AllMemoryFragmentation = new
List<MemoryFragmentation>();
```

```
  private ValidationMy _Validation = new ValidationMy();

  public MemoryFragmentationForm() {
    InitializeComponent();
    TestPBar.Minimum = 0; // Мінімальне значення прогрес бару
  }

  private void AddNewElementBtn_Click(object sender, EventArgs e) {
    if (IsDataEnteringCorrecting()) {
      MemoryFragmentation oneMemoryFragmentation = new MemoryFragmentation();
      oneMemoryFragmentation.NumFragments = Convert.ToInt32(NumFragmentsTBox.Text);
      oneMemoryFragmentation.FragmentSize = Convert.ToInt32(FragmentSizeTBox.Text);
      _AllMemoryFragmentation.Add(oneMemoryFragmentation);
      LoadDataInLargeArrayGW(_AllMemoryFragmentation);
      CliarData();
    }
  }

  private void TestBtn_Click(object sender, EventArgs e) {
    if (_AllMemoryFragmentation.Count >= 2) {
      // Очищаємо логи перед початком нових експериментів
      LogsTBox.Clear();

      // Ініціалізуємо графік перед проведенням експериментів
      InitializeChart();

      // Проведення серії експериментів
      PerformExperiments(_AllMemoryFragmentation);

      // Оновлюємо дані графіка після завершення експериментів
      UpdateChartWithData();
    } else {
      MessageBox.Show(NamesMy.Messages.InsufficientData, "Увага");
    }
  }

  private void MemoryFragmentationGridView_CellClick(object sender, DataGridViewCellEventArgs
e) {
    if (e.ColumnIndex == 3 && MemoryFragmentationGridView[0, e.RowIndex].Value.ToString() !=
_AllMemoryFragmentation[0].Message) {
      int num = Convert.ToInt32(MemoryFragmentationGridView[0, e.RowIndex].Value.ToString());
      for (int i = 0; i < _AllMemoryFragmentation.Count; i++) {
        if (num == _AllMemoryFragmentation[i].Number) {
          _AllMemoryFragmentation.RemoveAt(i);
          break;
        }
      }
      LoadDataInLargeArrayGW(_AllMemoryFragmentation);
    }
  }
```

```
// Метод для ініціалізації графіка
private void InitializeChart() {
  // Очищуємо серії та осі
  GraphicsCC.Series.Clear();
  GraphicsCC.AxisX.Clear();
  GraphicsCC.AxisY.Clear();

  // Створюємо серії для графіка
  var executionTimeSeries = new LineSeries {
    Title = "Час виконання",
    Values = new ChartValues<double>()
  };

  var memoryUsedSeries = new LineSeries {
    Title = "Використана пам'ять (MB)",
    Values = new ChartValues<double>()
  };

  // Додаємо серії на графік
  GraphicsCC.Series.Add(executionTimeSeries);
  GraphicsCC.Series.Add(memoryUsedSeries);

  // Додаємо підписи для осі X
  GraphicsCC.AxisX.Add(new LiveCharts.Wpf.Axis {
    Title = "Кількість фрагментів",
    Labels = _AllMemoryFragmentation.Select(ex => ex.NumFragments.ToString()).ToArray(),
    Separator = new LiveCharts.Wpf.Separator { Step = 1 }
  });

  // Вісь Y
  GraphicsCC.AxisY.Add(new LiveCharts.Wpf.Axis {
    Title = "Час виконання (мс)"
  });
}

// Метод для оновлення даних на графіку
private void UpdateChartWithData() {
  // Переконуємося, що є серії для оновлення
  if (GraphicsCC.Series.Count > 0) {
    var executionTimeSeries = (LineSeries)GraphicsCC.Series[0];
    var memoryUsedSeries = (LineSeries)GraphicsCC.Series[1];

    // Очищуємо старі дані
    executionTimeSeries.Values.Clear();
    memoryUsedSeries.Values.Clear();

    // Додаємо нові дані
    foreach (var experiment in _AllMemoryFragmentation) {
      executionTimeSeries.Values.Add(experiment.ExecutionTimeMilliseconds);
      memoryUsedSeries.Values.Add(experiment.MemoryUsedMB);
    }
  }
```

```
    }


    private void CliarData() {
      NumFragmentsTBox.Text = "0";
      FragmentSizeTBox.Text = "0";
    }


    // Метод для вимірювання операцій і збереження результатів у MemoryFragmentation
    private void MeasureMemoryFragmentation(MemoryFragmentation experiment) {
      PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time",
"_Total");
      PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
      Stopwatch stopwatch = new Stopwatch();
      stopwatch.Start();
      // Початкові виміри пам'яті та ЦП
      long initialMemory = GC.GetTotalMemory(true);
      double initialCpuUsage = cpuCounter.NextValue();
      // Список для зберігання фрагментованих блоків
      List<byte[]> fragmentedBlocks = new List<byte[]>();
      Random random = new Random();
      // Створення фрагментованої пам'яті
      for (int i = 0; i < experiment.NumFragments; i++) {
        // Визначення мінімального розміру блоку для запобігання помилкам
        int fragmentSize = experiment.FragmentSize < 1024 ? random.Next(1, experiment.FragmentSize)
          : random.Next(1024, experiment.FragmentSize);
        byte[] block = new byte[fragmentSize];
        // Ініціалізація масиву
        for (int j = 0; j < block.Length; j++) {
          block[j] = (byte)(j % 256);
        }
        // Додаємо блок у список
        fragmentedBlocks.Add(block);
        // Створення фрагментації шляхом випадкового звільнення частини блоків
        if (i % 3 == 0 && fragmentedBlocks.Count > 0) {
          fragmentedBlocks.RemoveAt(random.Next(fragmentedBlocks.Count)); // Видаляємо випадковий
блок
          GC.Collect(); // Примусова збірка сміття
        }
      }
      stopwatch.Stop();
      // Кінцеві виміри пам'яті та ЦП
      long finalMemory = GC.GetTotalMemory(false);
      double finalCpuUsage = cpuCounter.NextValue();
      double availableMemoryMB = ramCounter.NextValue();
      // Збереження результатів
      experiment.ExecutionTimeMilliseconds = stopwatch.Elapsed.TotalMilliseconds;
      experiment.MemoryUsedMB = (finalMemory - initialMemory) / (1024 * 1024);
      experiment.AverageCpuUsagePercent = (initialCpuUsage + finalCpuUsage) / 2;
      experiment.AvailableMemoryMB = availableMemoryMB;
```

```
      // Вивільнення пам'яті
      fragmentedBlocks.Clear();
      GC.Collect(); // Примусове очищення пам'яті
    }


    // Метод для відображення результатів у LogsTBox
    private void DisplayResults(ExperimentResult result) {
      // Виведення результатів у LogsTBox
      LogsTBox.AppendText($"Час виконання: {result.ExecutionTimeMilliseconds}
мс{Environment.NewLine}");
      LogsTBox.AppendText($"Використано пам'яті: {result.MemoryUsedMB}
MB{Environment.NewLine}");
      LogsTBox.AppendText($"Середнє навантаження процесора:
{result.AverageCpuUsagePercent}%{Environment.NewLine}");
      LogsTBox.AppendText($"Доступна пам'ять: {result.AvailableMemoryMB}
MB{Environment.NewLine}");
      Application.DoEvents();
    }

    // Метод для виконання серії експериментів з оновленням прогрес бару
    private void PerformExperiments(List<MemoryFragmentation> experiments) {
      TestPBar.Maximum = experiments.Count; // Встановлюємо максимальне значення для прогрес-
бару
      TestPBar.Value = 0; // Скидаємо прогрес на початок

      foreach (var experiment in experiments) {
        // Виведення номера експерименту та повідомлення перед запуском
        LogsTBox.AppendText($"Експеримент {experiment.Number}:
{experiment.Message}{Environment.NewLine}");

        // Проведення заміру і збереження результатів
        MeasureMemoryFragmentation(experiment);

        // Виведення результатів після завершення кожного експерименту
        DisplayResults(experiment);

        // Оновлюємо прогрес бар
        TestPBar.Value += 1;

        // Оновлюємо інтерфейс, щоб прогрес бар відображався під час виконання експериментів
        Application.DoEvents();
      }
    }
    private void LoadDataInLargeArrayGW(List<MemoryFragmentation> AllMemoryFragmentation) {
      ChangeNumber();
      MemoryFragmentationGridView.DataSource = null;
      MemoryFragmentationGridView.Columns.Clear();
      MemoryFragmentationGridView.AutoGenerateColumns = false;
      MemoryFragmentationGridView.RowHeadersVisible = false;

      MemoryFragmentationGridView.DataSource = AllMemoryFragmentation;
```

```
      if (AllMemoryFragmentation.Count > 0) {
        DataGridViewColumn numberColumn = new DataGridViewTextBoxColumn();
        numberColumn.HeaderText = "№ з/п";
        numberColumn.DataPropertyName = "Number";
        numberColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
        numberColumn.Width = NamesMy.SizeOptins.NumberSize;
        MemoryFragmentationGridView.Columns.Add(numberColumn);

        DataGridViewColumn NumFragmentsColumn = new DataGridViewTextBoxColumn();
        NumFragmentsColumn.HeaderText = "К-сть фрагментів";
        NumFragmentsColumn.DataPropertyName = "NumFragments";
        NumFragmentsColumn.Name = "NumFragments";
        NumFragmentsColumn.DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleRight;
        NumFragmentsColumn.Width = 120;
        MemoryFragmentationGridView.Columns.Add(NumFragmentsColumn);

        DataGridViewColumn FragmentSizeColumn = new DataGridViewTextBoxColumn();
        FragmentSizeColumn.HeaderText = "Макс. к-сть блоків";
        FragmentSizeColumn.DataPropertyName = "FragmentSize";
        FragmentSizeColumn.Name = "FragmentSize";
        FragmentSizeColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
        FragmentSizeColumn.Width = 120;
        MemoryFragmentationGridView.Columns.Add(FragmentSizeColumn);

        DataGridViewButtonColumn DeleteBtn = new DataGridViewButtonColumn();
        DeleteBtn.Text = "Видалити";
        DeleteBtn.UseColumnTextForButtonValue = true;
        DeleteBtn.ToolTipText = "Видалити";
        DeleteBtn.Width = NamesMy.SizeOptins.DeleteBtnSize;
        MemoryFragmentationGridView.Columns.Add(DeleteBtn);

        for (int i = 0; i < MemoryFragmentationGridView.Columns.Count; i++) {
          MemoryFragmentationGridView.Columns[i].HeaderCell.Style.BackColor = Color.LightGray;
        }
      }
    }

    private void ChangeNumber() {
      for (int i = 0; i < _AllMemoryFragmentation.Count; i++) {
        _AllMemoryFragmentation[i].Number = i + 1;
      }
    }

    private bool IsDataEnteringCorrecting() {
      bool isCorrect = true;
      if (_Validation.IsDataConvertToInt(NumFragmentsTBox.Text)) {
        if (_Validation.IsDataInThisScope(1, 10000000, Convert.ToInt32(FragmentSizeTBox.Text))) {
          NumFragmentsValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
        } else {
          NumFragmentsValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
```

```
        isCorrect = false;
       }
      } else {
       NumFragmentsValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
       isCorrect = false;
      }
      if (_Validation.IsDataConvertToInt(FragmentSizeTBox.Text)) {
       if (_Validation.IsDataInThisScope(1, 10000000, Convert.ToInt32(FragmentSizeTBox.Text))) {
        FragmentSizeValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
       } else {
        FragmentSizeValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
        isCorrect = false;
       }
      } else {
       FragmentSizeValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
       isCorrect = false;
      }
      return isCorrect;
     }


   }
}
```

Лістинг 6. Код класу «SelectionLargePileForm»

```
using DinMemWinApp.AppCode;
using DinMemWinApp.Classes;
using LiveCharts.Wpf;
using LiveCharts;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace DinMemWinApp.Forms.Operations {
  public partial class SelectionLargePileForm : Form {
    private List<LargeObject> _AllLargeObject = new List<LargeObject>();
    private ValidationMy _Validation = new ValidationMy();

    public SelectionLargePileForm() {
      InitializeComponent();
      TestPBar.Minimum = 0; // Мінімальне значення прогрес бару
    }

    private void AddNewElementBtn_Click(object sender, EventArgs e) {
      if (IsDataEnteringCorrecting()) {
```

```
        LargeObject oneLargeObject = new LargeObject();
        oneLargeObject.NumAllocations = Convert.ToInt32(NumAllocationsTBox.Text);
        oneLargeObject.ArraySize = Convert.ToInt32(ArraySizeTBox.Text);
        _AllLargeObject.Add(oneLargeObject);
        LoadDataInLargeArrayGW(_AllLargeObject);
        CliarData();
      }
    }

    private void TestBtn_Click(object sender, EventArgs e) {
      if (_AllLargeObject.Count >= 2) {
        // Очищаємо логи перед початком нових експериментів
        LogsTBox.Clear();

        // Ініціалізуємо графік перед проведенням експериментів
        InitializeChart();

        // Проведення серії експериментів
        PerformExperimentsWithLargeObjects(_AllLargeObject);

        // Оновлюємо дані графіка після завершення експериментів
        UpdateChartWithData();
      } else {
        MessageBox.Show(NamesMy.Messages.InsufficientData, "Увага");
      }
    }

    private void LargeObjectGridView_CellClick(object sender, DataGridViewCellEventArgs e) {
      if (e.ColumnIndex == 3 && LargeObjectGridView[0, e.RowIndex].Value.ToString() !=
_AllLargeObject[0].Message) {
        int num = Convert.ToInt32(LargeObjectGridView[0, e.RowIndex].Value.ToString());
        for (int i = 0; i < _AllLargeObject.Count; i++) {
          if (num == _AllLargeObject[i].Number) {
            _AllLargeObject.RemoveAt(i);
            break;
          }
        }
        LoadDataInLargeArrayGW(_AllLargeObject);
      }
    }

    // Метод для ініціалізації графіка
    private void InitializeChart() {
      // Очищуємо серії та осі
      GraphicsCC.Series.Clear();
      GraphicsCC.AxisX.Clear();
      GraphicsCC.AxisY.Clear();

      // Створюємо серії для графіка
      var executionTimeSeries = new LineSeries {
        Title = "Час виконання",
        Values = new ChartValues<double>()
```

```
  };

  var memoryUsedSeries = new LineSeries {
    Title = "Використана пам'ять (MB)",
    Values = new ChartValues<double>()
  };

  // Додаємо серії на графік
  GraphicsCC.Series.Add(executionTimeSeries);
  GraphicsCC.Series.Add(memoryUsedSeries);

  // Додаємо підписи для осі X
  GraphicsCC.AxisX.Add(new LiveCharts.Wpf.Axis {
    Title = "Кількість об'єктів",
    Labels = _AllLargeObject.Select(ex => ex.ArraySize.ToString()).ToArray(),
    Separator = new LiveCharts.Wpf.Separator { Step = 1 }
  });

  // Вісь Y
  GraphicsCC.AxisY.Add(new LiveCharts.Wpf.Axis {
    Title = "Час виконання (мс)"
  });
}

// Метод для оновлення даних на графіку
private void UpdateChartWithData() {
  // Переконуємося, що є серії для оновлення
  if (GraphicsCC.Series.Count > 0) {
    var executionTimeSeries = (LineSeries)GraphicsCC.Series[0];
    var memoryUsedSeries = (LineSeries)GraphicsCC.Series[1];

    // Очищуємо старі дані
    executionTimeSeries.Values.Clear();
    memoryUsedSeries.Values.Clear();

    // Додаємо нові дані
    foreach (var experiment in _AllLargeObject) {
      executionTimeSeries.Values.Add(experiment.ExecutionTimeMilliseconds);
      memoryUsedSeries.Values.Add(experiment.MemoryUsedMB);
    }
  }
}



private void CliarData() {
  NumAllocationsTBox.Text = "0";
  ArraySizeTBox.Text = "0";
}


// Метод для вимірювання операцій з великими об'єктами
```

```
    private void MeasureLargeObjectAllocation(LargeObject experiment) {
      PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time",
"_Total");
      PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
      Stopwatch stopwatch = new Stopwatch();
      stopwatch.Start();
      long initialMemory = GC.GetTotalMemory(true);
      double initialCpuUsage = cpuCounter.NextValue();
      // Список для великих об'єктів
      List<byte[]> largeObjects = new List<byte[]>();
      // Виділення пам'яті на великий масив байтів
      for (int i = 0; i < experiment.NumAllocations; i++) {
        // Виділення пам'яті для масиву байтів
        byte[] largeArray = new byte[experiment.ArraySize];
        // Ініціалізація масиву
        for (int j = 0; j < largeArray.Length; j++) {
          largeArray[j] = (byte)(j % 256); // Заповнення масиву байтами
        }
        // Додаємо масив до списку для збереження
        largeObjects.Add(largeArray);
      }
      stopwatch.Stop();
      long finalMemory = GC.GetTotalMemory(false);
      double finalCpuUsage = cpuCounter.NextValue();
      double availableMemoryMB = ramCounter.NextValue();
      experiment.ExecutionTimeMilliseconds = stopwatch.Elapsed.TotalMilliseconds;
      experiment.MemoryUsedMB = (finalMemory - initialMemory) / (1024 * 1024);
      experiment.AverageCpuUsagePercent = (initialCpuUsage + finalCpuUsage) / 2;
      experiment.AvailableMemoryMB = availableMemoryMB;
    }


    // Метод для відображення результатів у LogsTBox
    private void DisplayResults(ExperimentResult result) {
      // Виведення результатів у LogsTBox замість Console
      LogsTBox.AppendText($"Час виконання: {result.ExecutionTimeMilliseconds}
мс{Environment.NewLine}");
      LogsTBox.AppendText($"Використано пам'яті: {result.MemoryUsedMB}
MB{Environment.NewLine}");
      LogsTBox.AppendText($"Середнє навантаження процесора:
{result.AverageCpuUsagePercent}%{Environment.NewLine}");
      LogsTBox.AppendText($"Доступна пам'ять: {result.AvailableMemoryMB}
MB{Environment.NewLine}");
      Application.DoEvents();
    }

    //Метод для виконання серії експериментів
    private void PerformExperimentsWithLargeObjects(List<LargeObject> experiments) {
      TestPBar.Maximum = experiments.Count; // Встановлюємо максимальне значення для прогрес-
бару
      TestPBar.Value = 0; // Скидаємо прогрес на початок
```

```
    foreach (var experiment in experiments) {
      // Виведення номера експерименту та повідомлення перед запуском
      LogsTBox.AppendText($"Експеримент {experiment.Number}:
{experiment.Message}{Environment.NewLine}");

      // Проведення заміру і збереження результатів
      MeasureLargeObjectAllocation(experiment);

      // Виведення результатів після завершення кожного експерименту
      DisplayResults(experiment);

      // Оновлення прогрес-бару
      TestPBar.Value += 1;
      Application.DoEvents(); // Оновлення інтерфейсу
    }
  }


  private void LoadDataInLargeArrayGW(List<LargeObject> AllLargeObject) {
    ChangeNumber();
    LargeObjectGridView.DataSource = null;
    LargeObjectGridView.Columns.Clear();
    LargeObjectGridView.AutoGenerateColumns = false;
    LargeObjectGridView.RowHeadersVisible = false;

    LargeObjectGridView.DataSource = AllLargeObject;

    if (AllLargeObject.Count > 0) {
      DataGridViewColumn numberColumn = new DataGridViewTextBoxColumn();
      numberColumn.HeaderText = "№ з/п";
      numberColumn.DataPropertyName = "Number";
      numberColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
      numberColumn.Width = NamesMy.SizeOptins.NumberSize;
      LargeObjectGridView.Columns.Add(numberColumn);

      DataGridViewColumn NumAllocationsColumn = new DataGridViewTextBoxColumn();
      NumAllocationsColumn.HeaderText = "К-сть виділень";
      NumAllocationsColumn.DataPropertyName = "NumAllocations";
      NumAllocationsColumn.Name = "NumAllocations";
      NumAllocationsColumn.DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleRight;
      NumAllocationsColumn.Width = 120;
      LargeObjectGridView.Columns.Add(NumAllocationsColumn);

      DataGridViewColumn ArraySizeColumn = new DataGridViewTextBoxColumn();
      ArraySizeColumn.HeaderText = "К-сть об'єктів";
      ArraySizeColumn.DataPropertyName = "ArraySize";
      ArraySizeColumn.Name = "ArraySize";
      ArraySizeColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
      ArraySizeColumn.Width = 120;
      LargeObjectGridView.Columns.Add(ArraySizeColumn);
```

```
      DataGridViewButtonColumn DeleteBtn = new DataGridViewButtonColumn();
      DeleteBtn.Text = "Видалити";
      DeleteBtn.UseColumnTextForButtonValue = true;
      DeleteBtn.ToolTipText = "Видалити";
      DeleteBtn.Width = NamesMy.SizeOptins.DeleteBtnSize;
      LargeObjectGridView.Columns.Add(DeleteBtn);

      for (int i = 0; i < LargeObjectGridView.Columns.Count; i++) {
        LargeObjectGridView.Columns[i].HeaderCell.Style.BackColor = Color.LightGray;
      }
    }
  }

  private void ChangeNumber() {
    for (int i = 0; i < _AllLargeObject.Count; i++) {
      _AllLargeObject[i].Number = i + 1;
    }
  }

  private bool IsDataEnteringCorrecting() {
    bool isCorrect = true;
    if (_Validation.IsDataConvertToInt(NumAllocationsTBox.Text)) {
      if (_Validation.IsDataInThisScope(1, 10000000, Convert.ToInt32(NumAllocationsTBox.Text))) {
        NumAllocationsValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
      } else {
        NumAllocationsValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
        isCorrect = false;
      }
    } else {
      NumAllocationsValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
      isCorrect = false;
    }
    if (_Validation.IsDataConvertToInt(ArraySizeTBox.Text)) {
      if (_Validation.IsDataInThisScope(1, 10000000, Convert.ToInt32(ArraySizeTBox.Text))) {
        ArraySizeValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
      } else {
        ArraySizeValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
        isCorrect = false;
      }
    } else {
      ArraySizeValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
      isCorrect = false;
    }
    return isCorrect;
  }


 }
}
```

Лістинг 7. Код класу «StringManipulationsForm»

```
using DinMemWinApp.AppCode;
```

```
using DinMemWinApp.Classes;
using LiveCharts.Wpf;
using LiveCharts;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace DinMemWinApp.Forms.Operations {
  public partial class StringManipulationsForm : Form {
    private List<StringManipulations> _AllStringManipulations = new List<StringManipulations>();
    private ValidationMy _Validation = new ValidationMy();

    public StringManipulationsForm() {
      InitializeComponent();
      TestPBar.Minimum = 0; // Мінімальне значення прогрес бару
    }

    private void AddNewElementBtn_Click(object sender, EventArgs e) {
      if (IsDataEnteringCorrecting()) {
        StringManipulations oneStringManipulations = new StringManipulations();
        oneStringManipulations.OperationCount = Convert.ToInt32(OperationCountTBox.Text);
        oneStringManipulations.EnterString = EnterStringTBox.Text;
        _AllStringManipulations.Add(oneStringManipulations);
        LoadDataInLargeArrayGW(_AllStringManipulations);
        CliarData();
      }
    }

    private void TestBtn_Click(object sender, EventArgs e) {
      if (_AllStringManipulations.Count >= 2) {
        // Очищаємо логи перед початком нових експериментів
        LogsTBox.Clear();

        // Ініціалізуємо графік перед проведенням експериментів
        InitializeChart();

        // Проведення серії експериментів
        PerformExperimentsWithStringManipulations(_AllStringManipulations);

        // Оновлюємо дані графіка після завершення експериментів
        UpdateChartWithData();
      } else {
        MessageBox.Show(NamesMy.Messages.InsufficientData, "Увага");
      }
    }
```

```
  private void StringManipulationsGridView_CellClick(object sender, DataGridViewCellEventArgs e) {
    if (e.ColumnIndex == 3 && StringManipulationsGridView[0, e.RowIndex].Value.ToString() !=
_AllStringManipulations[0].Message) {
      int num = Convert.ToInt32(StringManipulationsGridView[0, e.RowIndex].Value.ToString());
      for (int i = 0; i < _AllStringManipulations.Count; i++) {
        if (num == _AllStringManipulations[i].Number) {
          _AllStringManipulations.RemoveAt(i);
          break;
        }
      }
      LoadDataInLargeArrayGW(_AllStringManipulations);
    }
  }



  // Метод для ініціалізації графіка
  private void InitializeChart() {
    // Очищуємо серії та осі
    GraphicsCC.Series.Clear();
    GraphicsCC.AxisX.Clear();
    GraphicsCC.AxisY.Clear();

    // Створюємо серії для графіка
    var executionTimeSeries = new LineSeries {
      Title = "Час виконання",
      Values = new ChartValues<double>()
    };

    var memoryUsedSeries = new LineSeries {
      Title = "Використана пам'ять (MB)",
      Values = new ChartValues<double>()
    };

    // Додаємо серії на графік
    GraphicsCC.Series.Add(executionTimeSeries);
    GraphicsCC.Series.Add(memoryUsedSeries);

    // Додаємо підписи для осі X
    GraphicsCC.AxisX.Add(new LiveCharts.Wpf.Axis {
      Title = "Кількість операцій",
      Labels = _AllStringManipulations.Select(ex => ex.OperationCount.ToString()).ToArray(),
      Separator = new LiveCharts.Wpf.Separator { Step = 1 }
    });

    // Вісь Y
    GraphicsCC.AxisY.Add(new LiveCharts.Wpf.Axis {
      Title = "Час виконання (мс)"
    });
  }
```

```
    // Метод для оновлення даних на графіку
    private void UpdateChartWithData() {
      // Переконуємося, що є серії для оновлення
      if (GraphicsCC.Series.Count > 0) {
        var executionTimeSeries = (LineSeries)GraphicsCC.Series[0];
        var memoryUsedSeries = (LineSeries)GraphicsCC.Series[1];

        // Очищуємо старі дані
        executionTimeSeries.Values.Clear();
        memoryUsedSeries.Values.Clear();

        // Додаємо нові дані
        foreach (var experiment in _AllStringManipulations) {
          executionTimeSeries.Values.Add(experiment.ExecutionTimeMilliseconds);
          memoryUsedSeries.Values.Add(experiment.MemoryUsedMB);
        }
      }
    }



    private void CliarData() {
      OperationCountTBox.Text = "0";
      EnterStringTBox.Text = "";
    }


    // Метод для вимірювання операцій з рядками
    private void MeasureStringManipulation(StringManipulations experiment) {
      PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time",
"_Total");
      PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
      Stopwatch stopwatch = new Stopwatch();
      stopwatch.Start();
      long initialMemory = GC.GetTotalMemory(true);
      double initialCpuUsage = cpuCounter.NextValue();
      // Операції з рядками
      string result = experiment.EnterString;
      List<string> generatedStrings = new List<string>(); // Збереження нових рядків
      // Для кожного символу генеруємо кілька нових рядків, змінюючи символ
      for (int i = 0; i < experiment.EnterString.Length; i++) {
        char[] charArray = result.ToCharArray();
        // Створюємо нову варіацію, змінюючи символ на індексі
        for (int j = 0; j < experiment.OperationCount; j++) {
          // Генеруємо новий символ на базі існуючого
          charArray[i] = (char)((charArray[i] + j) % 256); // Шифрування через циклічну зміну ASCII

          // Створюємо новий рядок і додаємо до списку
          string newString = new string(charArray);
          generatedStrings.Add(newString);
        }
      }
```

```
    // Цей процес лінійно збільшує використання пам'яті та час в залежності від довжини
EnterString і OperationCount

    stopwatch.Stop();

    long finalMemory = GC.GetTotalMemory(false);
    double finalCpuUsage = cpuCounter.NextValue();
    double availableMemoryMB = ramCounter.NextValue();

    experiment.ExecutionTimeMilliseconds = stopwatch.Elapsed.TotalMilliseconds;
    experiment.MemoryUsedMB = (finalMemory - initialMemory) / (1024 * 1024);
    experiment.AverageCpuUsagePercent = (initialCpuUsage + finalCpuUsage) / 2;
    experiment.AvailableMemoryMB = availableMemoryMB;
  }


  // Метод для відображення результатів у LogsTBox
  private void DisplayResults(ExperimentResult result) {
    LogsTBox.AppendText($"Час виконання: {result.ExecutionTimeMilliseconds}
мс{Environment.NewLine}");
    LogsTBox.AppendText($"Використано пам'яті: {result.MemoryUsedMB}
MB{Environment.NewLine}");
    LogsTBox.AppendText($"Середнє навантаження процесора:
{result.AverageCpuUsagePercent}%{Environment.NewLine}");
    LogsTBox.AppendText($"Доступна пам'ять: {result.AvailableMemoryMB}
MB{Environment.NewLine}");
    Application.DoEvents();
  }


  // Метод для виконання серії експериментів з оновленням прогрес бару
  private void PerformExperimentsWithStringManipulations(List<StringManipulations> experiments) {
    TestPBar.Maximum = experiments.Count; // Встановлюємо максимальне значення для прогрес
бару
    TestPBar.Value = 0; // Скидаємо прогрес на початок

    foreach (var experiment in experiments) {
      // Виведення номера експерименту та повідомлення перед запуском
      LogsTBox.AppendText($"Експеримент {experiment.Number}:
{experiment.Message}{Environment.NewLine}");

      // Проведення заміру і збереження результатів
      MeasureStringManipulation(experiment);

      // Виведення результатів після завершення кожного експерименту
      DisplayResults(experiment);

      // Оновлення прогрес-бару
      TestPBar.Value += 1;
      Application.DoEvents(); // Оновлення інтерфейсу
    }
  }
```

```
    private void LoadDataInLargeArrayGW(List<StringManipulations> AllStringManipulations) {
      ChangeNumber();
      StringManipulationsGridView.DataSource = null;
      StringManipulationsGridView.Columns.Clear();
      StringManipulationsGridView.AutoGenerateColumns = false;
      StringManipulationsGridView.RowHeadersVisible = false;

      StringManipulationsGridView.DataSource = AllStringManipulations;

      if (AllStringManipulations.Count > 0) {
        DataGridViewColumn numberColumn = new DataGridViewTextBoxColumn();
        numberColumn.HeaderText = "№ з/п";
        numberColumn.DataPropertyName = "Number";
        numberColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
        numberColumn.Width = 50;
        StringManipulationsGridView.Columns.Add(numberColumn);

        DataGridViewColumn OperationCountColumn = new DataGridViewTextBoxColumn();
        OperationCountColumn.HeaderText = "К-сть операцій";
        OperationCountColumn.DataPropertyName = "OperationCount";
        OperationCountColumn.Name = "OperationCount";
        OperationCountColumn.DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleRight;
        OperationCountColumn.Width = 100;
        StringManipulationsGridView.Columns.Add(OperationCountColumn);

        DataGridViewColumn EnterStringColumn = new DataGridViewTextBoxColumn();
        EnterStringColumn.HeaderText = "Текст маніпуляції";
        EnterStringColumn.DataPropertyName = "EnterString";
        EnterStringColumn.Name = "EnterString";
        EnterStringColumn.Width = 120;
        StringManipulationsGridView.Columns.Add(EnterStringColumn);

        DataGridViewButtonColumn DeleteBtn = new DataGridViewButtonColumn();
        DeleteBtn.Text = "Видалити";
        DeleteBtn.UseColumnTextForButtonValue = true;
        DeleteBtn.ToolTipText = "Видалити";
        DeleteBtn.Width = NamesMy.SizeOptins.DeleteBtnSize;
        StringManipulationsGridView.Columns.Add(DeleteBtn);

        for (int i = 0; i < StringManipulationsGridView.Columns.Count; i++) {
          StringManipulationsGridView.Columns[i].HeaderCell.Style.BackColor = Color.LightGray;
        }
      }
    }

    private void ChangeNumber() {
      for (int i = 0; i < _AllStringManipulations.Count; i++) {
        _AllStringManipulations[i].Number = i + 1;
      }
```

```
    }

    private bool IsDataEnteringCorrecting() {
      bool isCorrect = true;
      if (_Validation.IsDataConvertToInt(OperationCountTBox.Text)) {
        if (_Validation.IsDataInThisScope(1, 10000000, Convert.ToInt32(OperationCountTBox.Text))) {
          OperationCountValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
        } else {
          OperationCountValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
          isCorrect = false;
        }
      } else {
        OperationCountValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
        isCorrect = false;
      }
      if (_Validation.IsDataEntering(EnterStringTBox.Text)) {
          EnterStringValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
      } else {
        EnterStringValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
        isCorrect = false;
      }
      return isCorrect;
    }



  }
}
```

Лістинг 8. Код класу «WorkWithDictionariesForm»

```
using DinMemWinApp.AppCode;
using DinMemWinApp.Classes;
using LiveCharts.Wpf;
using LiveCharts;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace DinMemWinApp.Forms.Operations {
  public partial class WorkWithDictionariesForm : Form {
    private List<DictionarieOperation> _AllDictionarieOperation = new List<DictionarieOperation>();
    private ValidationMy _Validation = new ValidationMy();

    public WorkWithDictionariesForm() {
      InitializeComponent();
```

```
    TestPBar.Minimum = 0; // Мінімальне значення прогрес бару
  }

  private void AddNewElementBtn_Click(object sender, EventArgs e) {
    if (IsDataEnteringCorrecting()) {
      DictionarieOperation oneDictionarieOperation = new DictionarieOperation();
      oneDictionarieOperation.OperationCount = Convert.ToInt32(OperationCountTBox.Text);
      _AllDictionarieOperation.Add(oneDictionarieOperation);
      LoadDataInLargeArrayGW(_AllDictionarieOperation);
      CliarData();
    }
  }

  private void TestBtn_Click(object sender, EventArgs e) {
    if (_AllDictionarieOperation.Count >= 2) {
      // Очищаємо логи перед початком нових експериментів
      LogsTBox.Clear();

      // Ініціалізуємо графік перед проведенням експериментів
      InitializeChart();

      // Проведення серії експериментів
      PerformExperimentsWithDictionarieOperation(_AllDictionarieOperation);

      // Оновлюємо дані графіка після завершення експериментів
      UpdateChartWithData();
    } else {
      MessageBox.Show(NamesMy.Messages.InsufficientData, "Увага");
    }
  }

  private void DictionarieOperationGridView_CellClick(object sender, DataGridViewCellEventArgs e)
{
    if (e.ColumnIndex == 2 && DictionarieOperationGridView[0, e.RowIndex].Value.ToString() !=
_AllDictionarieOperation[0].Message) {
      int num = Convert.ToInt32(DictionarieOperationGridView[0, e.RowIndex].Value.ToString());
      for (int i = 0; i < _AllDictionarieOperation.Count; i++) {
        if (num == _AllDictionarieOperation[i].Number) {
          _AllDictionarieOperation.RemoveAt(i);
          break;
        }
      }
      LoadDataInLargeArrayGW(_AllDictionarieOperation);
    }
  }

  private void CliarData() {
    OperationCountTBox.Text = "0";
  }

  // Метод для ініціалізації графіка
  private void InitializeChart() {
```

```
  // Очищуємо серії та осі
  GraphicsCC.Series.Clear();
  GraphicsCC.AxisX.Clear();
  GraphicsCC.AxisY.Clear();

  // Створюємо серії для графіка
  var executionTimeSeries = new LineSeries {
    Title = "Час виконання",
    Values = new ChartValues<double>()
  };

  var memoryUsedSeries = new LineSeries {
    Title = "Використана пам'ять (MB)",
    Values = new ChartValues<double>()
  };

  // Додаємо серії на графік
  GraphicsCC.Series.Add(executionTimeSeries);
  GraphicsCC.Series.Add(memoryUsedSeries);

  // Додаємо підписи для осі X
  GraphicsCC.AxisX.Add(new LiveCharts.Wpf.Axis {
    Title = "Кількість елементів",
    Labels = _AllDictionarieOperation.Select(ex => ex.OperationCount.ToString()).ToArray(),
    Separator = new LiveCharts.Wpf.Separator { Step = 1 }
  });

  // Вісь Y
  GraphicsCC.AxisY.Add(new LiveCharts.Wpf.Axis {
    Title = "Час виконання (мс)"
  });
 }

 // Метод для оновлення даних на графіку
 private void UpdateChartWithData() {
  // Переконуємося, що є серії для оновлення
  if (GraphicsCC.Series.Count > 0) {
    var executionTimeSeries = (LineSeries)GraphicsCC.Series[0];
    var memoryUsedSeries = (LineSeries)GraphicsCC.Series[1];

    // Очищуємо старі дані
    executionTimeSeries.Values.Clear();
    memoryUsedSeries.Values.Clear();

    // Додаємо нові дані
    foreach (var experiment in _AllDictionarieOperation) {
      executionTimeSeries.Values.Add(experiment.ExecutionTimeMilliseconds);
      memoryUsedSeries.Values.Add(experiment.MemoryUsedMB);
    }
  }
 }
```

```
    // Метод для вимірювання операцій і збереження результатів у DynamicStruct
    private void MeasureDictionarieAllocation(DictionarieOperation experiment) {
      PerformanceCounter cpuCounter = new PerformanceCounter("Processor", "% Processor Time",
"_Total");
      PerformanceCounter ramCounter = new PerformanceCounter("Memory", "Available MBytes");
      Stopwatch stopwatch = new Stopwatch();
      stopwatch.Start();
      long initialMemory = GC.GetTotalMemory(true);
      double initialCpuUsage = cpuCounter.NextValue();
      // Ініціалізація словника
      Dictionary<int, int> dictionary = new Dictionary<int, int>();
      // Додавання елементів у словник
      for (int i = 0; i < experiment.OperationCount; i++) {
        dictionary[i] = i; // Додаємо елемент у словник
      }
      // Операції з елементами словника: доступ до елементів і видалення
      if (dictionary.Count > 0) {
        // Доступ до першого елемента
        int firstKey = dictionary.Keys.First();
        int firstValue = dictionary[firstKey];
        // Видалення останнього елемента
        int lastKey = dictionary.Keys.Last();
        dictionary.Remove(lastKey);
        // Очищення словника
        dictionary.Clear();
      }

      stopwatch.Stop();

      long finalMemory = GC.GetTotalMemory(false);
      double finalCpuUsage = cpuCounter.NextValue();
      double availableMemoryMB = ramCounter.NextValue();

      // Збереження результатів експерименту
      experiment.ExecutionTimeMilliseconds = stopwatch.Elapsed.TotalMilliseconds;
      experiment.MemoryUsedMB = (finalMemory - initialMemory) / (1024 * 1024);
      experiment.AverageCpuUsagePercent = (initialCpuUsage + finalCpuUsage) / 2;
      experiment.AvailableMemoryMB = availableMemoryMB;
    }


    // Метод для відображення результатів у текстовому боксі або в консолі
    private void DisplayResults(ExperimentResult result) {
      // Виведення результатів у LogsTBox або Console
      LogsTBox.AppendText($"Час виконання: {result.ExecutionTimeMilliseconds}
мс{Environment.NewLine}");
      LogsTBox.AppendText($"Використано пам'яті: {result.MemoryUsedMB}
MB{Environment.NewLine}");
      LogsTBox.AppendText($"Середнє навантаження процесора:
{result.AverageCpuUsagePercent}%{Environment.NewLine}");
      LogsTBox.AppendText($"Доступна пам'ять: {result.AvailableMemoryMB}
MB{Environment.NewLine}");
```

```
      Application.DoEvents();
    }

    // Метод для виконання серії експериментів з оновленням прогрес бару
    private void PerformExperimentsWithDictionarieOperation(List<DictionarieOperation> experiments)
{
      TestPBar.Maximum = experiments.Count; // Встановлюємо максимальне значення для прогрес-
бару
      TestPBar.Value = 0; // Скидаємо прогрес на початок

      foreach (var experiment in experiments) {
        // Виведення номера експерименту та повідомлення перед запуском
        LogsTBox.AppendText($"Експеримент {experiment.Number}:
{experiment.Message}{Environment.NewLine}");

        // Проведення заміру і збереження результатів
        MeasureDictionarieAllocation(experiment);

        // Виведення результатів після завершення кожного експерименту
        DisplayResults(experiment);

        // Оновлення прогрес-бару
        TestPBar.Value += 1;
        Application.DoEvents(); // Оновлення інтерфейсу
      }
    }


    private void LoadDataInLargeArrayGW(List<DictionarieOperation> AllDictionarieOperation) {
      ChangeNumber();
      DictionarieOperationGridView.DataSource = null;
      DictionarieOperationGridView.Columns.Clear();
      DictionarieOperationGridView.AutoGenerateColumns = false;
      DictionarieOperationGridView.RowHeadersVisible = false;

      DictionarieOperationGridView.DataSource = AllDictionarieOperation;

      if (AllDictionarieOperation.Count > 0) {
        DataGridViewColumn numberColumn = new DataGridViewTextBoxColumn();
        numberColumn.HeaderText = "№ з/п";
        numberColumn.DataPropertyName = "Number";
        numberColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
        numberColumn.Width = NamesMy.SizeOptins.NumberSize;
        DictionarieOperationGridView.Columns.Add(numberColumn);

        DataGridViewColumn OperationCountColumn = new DataGridViewTextBoxColumn();
        OperationCountColumn.HeaderText = "К-сть елементів словника";
        OperationCountColumn.DataPropertyName = "OperationCount";
        OperationCountColumn.Name = "OperationCount";
        OperationCountColumn.DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleRight;
        OperationCountColumn.Width = 120;
```

```
      DictionarieOperationGridView.Columns.Add(OperationCountColumn);

      DataGridViewButtonColumn DeleteBtn = new DataGridViewButtonColumn();
      DeleteBtn.Text = "Видалити";
      DeleteBtn.UseColumnTextForButtonValue = true;
      DeleteBtn.ToolTipText = "Видалити";
      DeleteBtn.Width = NamesMy.SizeOptins.DeleteBtnSize;
      DictionarieOperationGridView.Columns.Add(DeleteBtn);

      for (int i = 0; i < DictionarieOperationGridView.Columns.Count; i++) {
        DictionarieOperationGridView.Columns[i].HeaderCell.Style.BackColor = Color.LightGray;
      }
    }
  }

  private void ChangeNumber() {
    for (int i = 0; i < _AllDictionarieOperation.Count; i++) {
      _AllDictionarieOperation[i].Number = i + 1;
    }
  }

  private bool IsDataEnteringCorrecting() {
    bool isCorrect = true;
    if (_Validation.IsDataConvertToInt(OperationCountTBox.Text)) {
      if (_Validation.IsDataInThisScope(1, 10000000, Convert.ToInt32(OperationCountTBox.Text))) {
        DynamicStructValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
      } else {
        DynamicStructValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
        isCorrect = false;
      }
    } else {
      DynamicStructValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
      isCorrect = false;
    }
    return isCorrect;
  }


  }
}
```