Міністерство освіти України

Національний технічний університет "ХПІ"

кафедра "Інформатики та інтелектуальної власності"

Звіт

Індивідуальне дослідницьке завдання

з дисципліни "Кросплатформне програмування"

Виконав: студент групи КН-1224С

Мульков М. В.

Перевірив:

Івашко А.В.

Харків 2025

# Зміст

[Завдання 5](#_Toc198312655)

[Реалізація коду 8](#_Toc198312656)

[1. Робота з текстовою інформацією у .NET Framework 8](#_Toc198312657)

[Алгоритм коду 8](#_Toc198312658)

[Тестування коду 8](#_Toc198312659)

[2. Робота з рваними масивами у .NET Framework 9](#_Toc198312660)

[Алгоритм коду 9](#_Toc198312661)

[Тестування коду 10](#_Toc198312662)

[3. Робота з успадкуванням класів у .NET Framework 12](#_Toc198312663)

[Алгоритм коду 12](#_Toc198312664)

[Тестування коду 13](#_Toc198312665)

[4. Робота з класом-контейнером у .NET Framework 15](#_Toc198312666)

[Алгоритм коду 15](#_Toc198312667)

[Тестування коду 16](#_Toc198312668)

[5. Робота з індексаторами у класі-контейнері 18](#_Toc198312669)

[Алгоритм коду 18](#_Toc198312670)

[Тестування коду 20](#_Toc198312671)

[6. Робота з опрацюванням виняткових ситуацій у .NET Framework 22](#_Toc198312672)

[Алгоритм коду 22](#_Toc198312673)

[Тестування коду 23](#_Toc198312674)

[7. Робота з інтерфейсами у .NET Framework 25](#_Toc198312675)

[Алгоритм коду 25](#_Toc198312676)

[Тестування коду 26](#_Toc198312677)

[8. Робота з узагальненнями та реалізація різних типів контейнерів 28](#_Toc198312678)

[Алгоритм коду 28](#_Toc198312679)

[Тестування коду 30](#_Toc198312680)

[9. Реалізація ітераторів у контейнерах у .NET Framework 32](#_Toc198312681)

[Алгоритм коду 32](#_Toc198312682)

[Тестування коду 33](#_Toc198312683)

[10. Робота з іменованими ітераторами (генераторами) 36](#_Toc198312684)

[Алгоритм коду 36](#_Toc198312685)

[Тестування коду 37](#_Toc198312686)

[11. Реалізація власної бінарної серіалізації та десеріалізації контейнерів 39](#_Toc198312687)

[Алгоритм коду 39](#_Toc198312688)

[Тестування коду 41](#_Toc198312689)

[12. Використання делегатів для сортування та пошуку в узагальнених контейнерах 44](#_Toc198312690)

[Алгоритм коду 44](#_Toc198312691)

[Тестування коду 46](#_Toc198312692)

[13. Використання подій для динамічного відстеження сумарної вартості у контейнерах 48](#_Toc198312693)

[Алгоритм коду 48](#_Toc198312694)

[Тестування коду 50](#_Toc198312695)

[14. Використання LINQ для аналізу даних у контейнерах 55](#_Toc198312696)

[Алгоритм коду 55](#_Toc198312697)

[Тестування коду 56](#_Toc198312698)

[Висновок 58](#_Toc198312699)

[Додаток А 60](#_Toc198312700)

**Розробка системи управління каталогом товарів з використанням об'єктно-орієнтованого програмування, узагальнених контейнерів та LINQ у C#**

Мета роботи: Набути комплексних практичних навичок з об'єктно-орієнтованого програмування на мові C# шляхом розробки ієрархії класів для представлення товарів, створення власних узагальнених класів-контейнерів для їх зберігання та маніпуляції, реалізації механізмів сортування, пошуку, ітерації, обробки виняткових ситуацій, серіалізації/десеріалізації даних, динамічного відстеження сумарної вартості за допомогою подій та використання мови інтегрованих запитів LINQ для аналізу даних.

# Завдання

В рамках iндивідуального дослідницького завдання було поставлено та виконано наступні завдання:

1. Проектування та реалізація ієрархії класів товарів:

Створити інтерфейс IName, що успадковується від IComparable та вимагає наявності властивості Name.

Розробити базовий клас Product та кілька похідних класів для специфічної групи товарів, що відрізняються набором властивостей.

Реалізувати для кожного класу конструктори за замовчуванням та з параметрами.

Перевантажити метод ToString() для надання інформативного рядкового представлення об'єктів.

Забезпечити реалізацію інтерфейсу IName (включаючи метод CompareTo) у класах ієрархії.

Інтегрувати інтерфейс IPrice для класів, що мають ціну, та реалізувати подію PriceChanged для відстеження змін ціни.

1. Розробка узагальненого класу-контейнера Container<T> (на основі масиву):

Створити клас-узагальнення Container<T> з обмеженням на тип T (має реалізовувати IName).

Реалізувати функціонал додавання об'єкта в контейнер (з автоматичним розширенням внутрішнього масиву).

Реалізувати функціонал видалення об'єкта з контейнера за його поточним індексом.

Реалізувати метод Sort(Comparison<T> comparison) для впорядкування інформації у контейнері за допомогою переданого методу порівняння.

Реалізувати перетворення вмісту контейнера до рядка (метод ToString()).

Реалізувати індексатори для отримання об'єкта за порядковим номером додавання (ID вставки) та за ім'ям товару.

Забезпечити можливість ітерації по контейнеру за допомогою циклу foreach шляхом реалізації IEnumerable<T> та власного класу-ітератора IEnumerator<T> (без використання yield).

Реалізувати додатковий ітератор (з використанням yield), який повертає об'єкти, імена яких починаються на вказаний рядок (або містять підрядок).

Реалізувати методи Find(Predicate<T> match) та FindAll(Predicate<T> match) для пошуку об'єктів за заданою умовою.

Реалізувати механізм динамічного відстеження сумарної вартості товарів у контейнері, використовуючи підписку на подію PriceChanged об'єктів.

1. (Опціонально, якщо реалізовувався другий тип контейнера) Розробка узагальненого класу-контейнера ContainerLinkedList<T> (на основі двозв'язного списку):

Створити клас-узагальнення ContainerLinkedList<T> з аналогічними обмеженнями та базовим функціоналом, що й Container<T>, але використовуючи структуру двозв'язного списку.

1. Обробка виняткових ситуацій:

Створити власний клас винятку (наприклад, ValueLessThanZero) для обробки помилок, пов'язаних з логікою даних.

Інтегрувати генерацію та обробку стандартних та власного винятків у класи ієрархії та контейнери.

1. Серіалізація/десеріалізація:

Реалізувати власний механізм бінарної (або JSON, як було в останній версії коду) серіалізації/десеріалізації стану контейнера та об'єктів у ньому без використання стандартних форматерів .NET. Забезпечити запис та відновлення типу контейнера та типів його елементів.

1. Використання LINQ:

Реалізувати LINQ-запити для знаходження найдешевшого та найдорожчого товару в активному контейнері.

Реалізувати LINQ-запит для знаходження середньої вартості товарів для кожної категорії (типу класу).

# Реалізація коду

# Робота з текстовою інформацією у .NET Framework

## Алгоритм коду

Спочатку на екрані з'являється меню з двома варіантами: ввести речення для обробки або вийти з програми.

Якщо користувач вибирає перший варіант, програма запитує у користувача введення рядка тексту (речення).

Рядок, введений користувачем, розбивається на окремі слова за допомогою методу Split.

Для кожного слова перевіряється його довжина. Якщо довжина слова більше або дорівнює 3 символам, воно додається до списку.

Після обробки всі слова, довжина яких більше або дорівнює 3 символам, об'єднуються в один рядок через пробіл. Результат виводиться на екран.

Якщо користувач вибирає варіант виходу з програми, програма завершується.

В результаті на екран виводяться тільки ті слова, які мають довжину більше або дорівнюють 3 символам.

## 

## Тестування коду

Тестування роботи коду зображено на рис. 1.1.

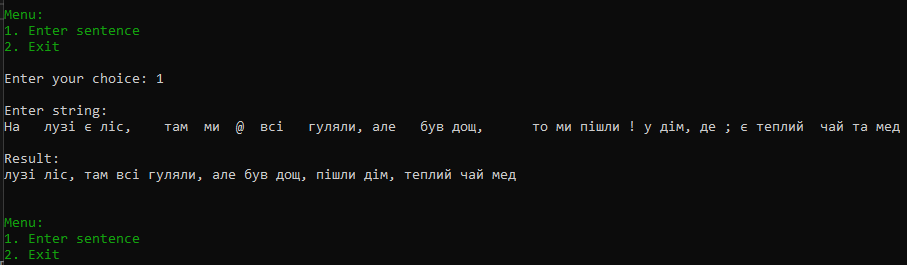


Рис. 1.1. Тестування коду

# Робота з рваними масивами у .NET Framework

## Алгоритм коду

Повний код програми відображено в додатку А.

1. Виведення меню

Програма запускається в безкінечному циклі, доки користувач не вибере вихід. На екран виводиться меню з трьома варіантами:

* Створити випадковий зубчастий масив.
* Ввести масив вручну.
* Вийти з програми.

Користувач вводить вибір, і залежно від введених даних виконується відповідний блок коду.

2. Створення випадкового масиву

Генерується випадкова кількість рядків від 1 до 7. Для кожного рядка генерується випадкова довжина від 0 до 7. Заповнюється випадковими числами в діапазоні від -7 до 7.

У результаті створюється зубчастий масив, у якому кожен рядок має різну довжину.

3. Ручне введення масиву

Користувач вводить кількість рядків. Для кожного рядка вводиться кількість елементів. Користувач вводить елементи, які розбиваються на окремі числа та записуються в масив.

4. Виведення початкового масиву

Викликається функція PrintArray, яка проходить по всіх рядках масиву та виводить їх на екран. Якщо рядок порожній, виводиться (empty). Якщо всі елементи рядка рівні нулю, вони виділяються червоним кольором і виводиться їх сума.

5. Видалення рядків, що складаються тільки з нулів

Викликається функція DeleteZeroRows. Спочатку підраховується кількість рядків, у яких є хоча б один ненульовий елемент. Потім створюється новий масив, у який записуються тільки ненульові рядки. Повертається новий масив без рядків, що містять лише нулі.

6. Виведення масиву після видалення нульових рядків

Викликається PrintArray, щоб показати змінений масив після видалення рядків із нулями.

7. Сортування рядків за зростанням суми елементів

Проходиться двома вкладеними циклами по всіх рядках масиву. Якщо сума елементів поточного рядка більша, ніж у наступного, вони міняються місцями. У результаті рядки масиву впорядковуються за зростанням їх суми.

8. Виведення відсортованого масиву

Знову викликається PrintArray, щоб показати відсортований масив.

9. Повернення до меню або вихід із програми

Після виведення масиву користувач повертається до головного меню. Якщо користувач вибирає "3", програма завершує виконання.

## Тестування коду

Тестування коду з автоматичним створенням масиву відображено на рис. 2.1.

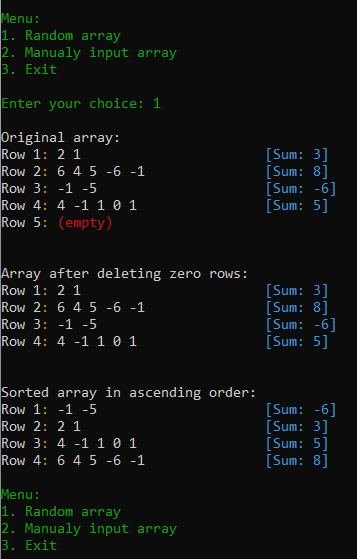


Рис. 2.1. Тестування коду з автоматичним створенням масиву

Тестування коду з ручним введенням масиву відображено на рис. 2.2.

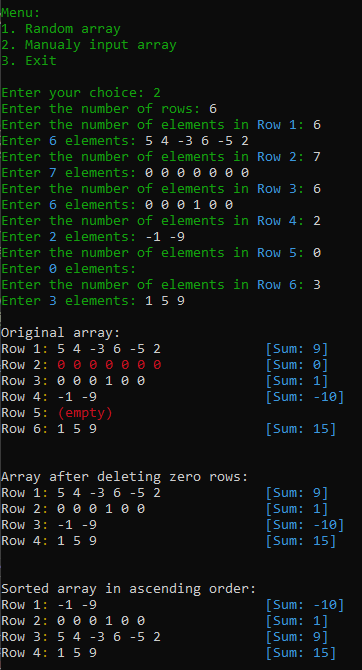


Рис. 2.2. Тестування коду з ручним введенням масиву

# Робота з успадкуванням класів у .NET Framework

## Алгоритм коду

Повний код програми відображено у додатку А.

1. Виведення головного меню

Після запуску програми користувачу відображається головне меню, оформлене кольоровими шрифтами.

Меню містить три варіанти: тест з випадковим створенням об’єктів, тест з ручним введенням параметрів, а також опцію для виходу з програми. Користувач вводить свій вибір, і програма перевіряє коректність введення. У разі некоректного вибору повідомлення про помилку виводиться червоним кольором.

Програма працює в безкінечному циклі, що дозволяє користувачу повторювати тестування або переходити між режимами. Вибір опції «Вихід» припиняє виконання програми.

2. Режим автоматичного створення об’єктів

При виборі цього режиму програма переходить до етапу генерації параметрів.

Використовуються заздалегідь визначені масиви, що містять типові значення для назв, локацій, типів товарів, інвестицій та ґрунтів. За допомогою генератора випадкових чисел програма обирає значення із цих масивів.

Для кожного класу (Product, RealEstate, RealEstateInvestment, Apartment, House, Hotel, LandPlot) створюється об’єкт із випадковим набором параметрів. Наприклад, для класу Product генерується назва з масиву та випадкова вартість у визначеному діапазоні.

Після створення кожного об’єкту його інформація виводиться на екран із застосуванням відповідного кольору, що робить вивід наочним. Межі між блоками інформації відокремлюються додатковими порожніми рядками.

3. Режим ручного введення

При виборі цього режиму користувачу пропонується окреме меню з переліком типів об’єктів, які можна створити. Після вибору типу об’єкту користувача покроково запрошують ввести необхідні дані (наприклад, назву, ціну, локацію, розмір, тип та інші параметри залежно від обраного класу).

Після введення даних створюється відповідний об’єкт, і його інформація виводиться на екран у кольоровому форматі. Для кожного типу об’єкта використовується окремий колір, що дозволяє швидко орієнтуватися у результатах тестування.

Програма після завершення ручного введення повертає користувача до головного меню.

## Тестування коду

Тестування коду з автоматичним створенням відображено на рис. 3.1.



Рис. 3.1. Тестування коду з автоматичним створенням

Тестування коду з ручним введенням відображено на рис. 3.2.

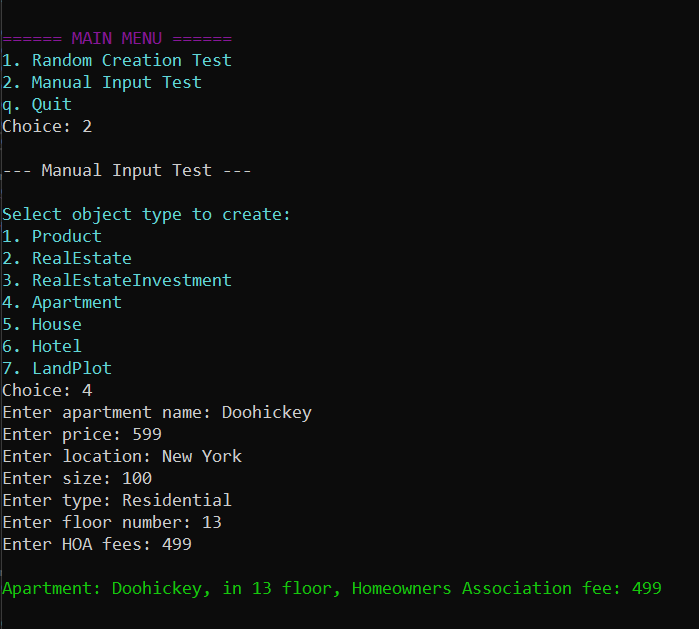


Рис. 3.2. Тестування коду з ручним введенням

# Робота з класом-контейнером у .NET Framework

## Алгоритм коду

Повний код програми відображено у додатку А.

1. Створення класу Container

Клас Container реалізовано як оболонка над масивом items типу object[]. Приватні поля count та size використовуються для відстеження поточної кількості елементів та розміру масиву відповідно. Конструктор за замовчуванням ініціалізує масив початковим розміром в 1 елемент.

2. Реалізація методу Add()

Метод Add відповідає за додавання нового об'єкта до контейнера. Перед додаванням перевіряється, чи є місце у внутрішньому масиві (count == size). Якщо масив заповнений, створюється новий масив подвійного розміру, елементи зі старого масиву копіюються до нового, після чого внутрішній масив items посилається на цей новий, збільшений масив. Новий об'єкт додається в кінець масиву, а лічильник count інкрементується.

3. Реалізація методу RemoveById()

Метод RemoveById видаляє об'єкт за вказаним індексом \_index. Спочатку виконується перевірка на коректність індексу. Якщо індекс виходить за межі масиву, генерується виняток IndexOutOfRangeException. Для видалення елемента створюється новий масив newArray того ж розміру. Елементи з вихідного масиву копіюються до нового масиву, пропускаючи елемент за вказаним індексом. Масив items замінюється новим масивом, а count зменшується. Метод повертає видалений об'єкт.

4. Реалізація методу Sort()

Метод Sort реалізує сортування об'єктів у контейнері за властивістю Price. Використано алгоритм бульбашкового сортування. Для порівняння цін використовується допоміжний приватний метод GetPrice(object item), який за допомогою рефлексії отримує значення властивості "Price" об'єкта, якщо така властивість існує та має тип decimal. У випадку помилки під час отримання ціни, виводиться повідомлення про помилку в консоль.

5. Реалізація методу ToString()

Метод ToString перетворює вміст контейнера в рядок. Він ітерується по масиву items та викликає метод ToString() для кожного об'єкта, додаючи результат до результуючого рядка з символом нового рядка.

6. Консольна програма тестування

Консольна програма надає користувачу інтерактивне меню для тестування класу Container. Реалізовано режими автоматичної та ручної генерації об'єктів для додавання в контейнер, перегляд вмісту контейнера, сортування та видалення елементів за індексом. Для наочності виведення інформації використовується кольорове оформлення консолі. Для виведення вмісту контейнера реалізовано табличне представлення з пагінацією для зручного перегляду великої кількості об'єктів.

## Тестування коду

Тестування коду з автоматичним створенням відображено на рис. 4.1.

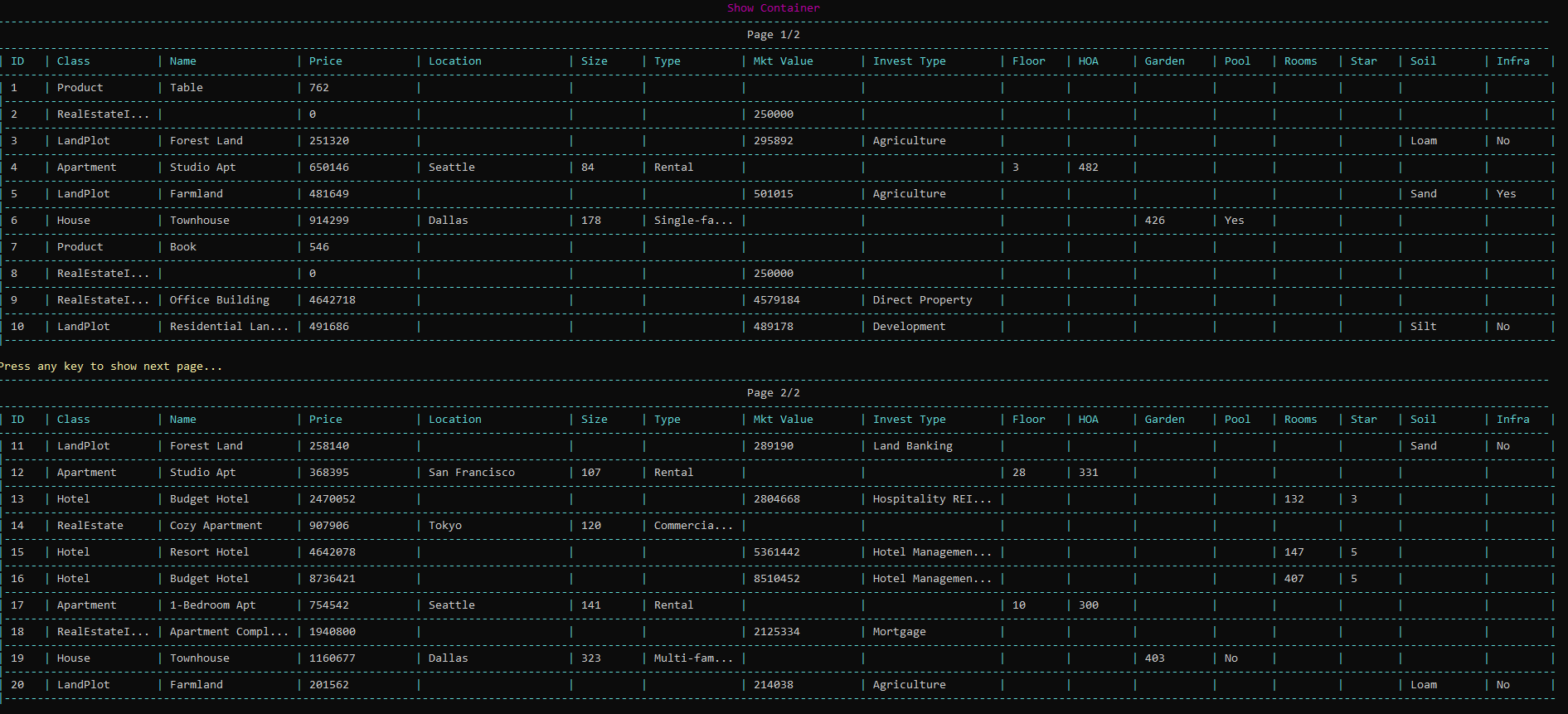


Рис. 4.1. Тестування коду з автоматичним створенням

Тестування коду сортування за цiною відображено на рис. 4.2.

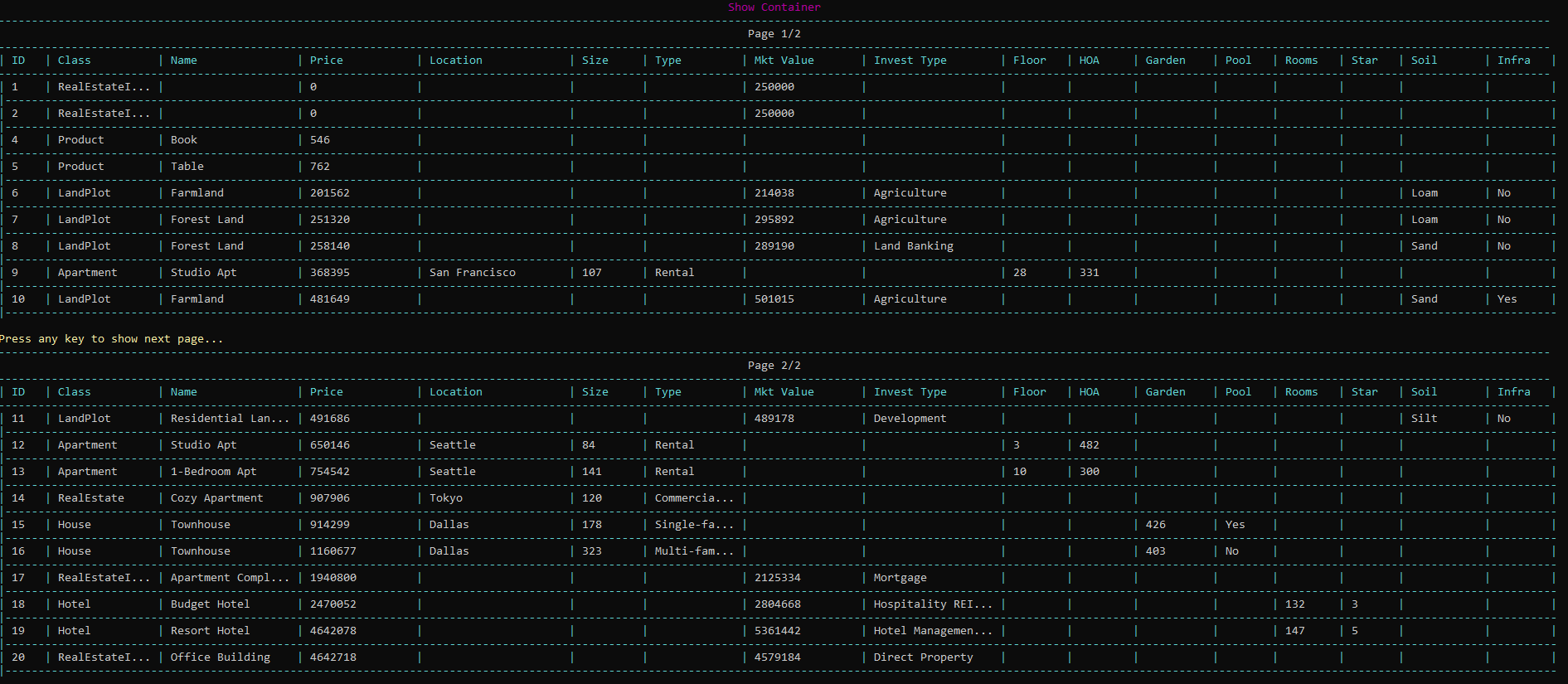


Рис. 4.2. Тестування коду сортування

Тестування коду видалення за iндексом відображено на рис. 4.3.

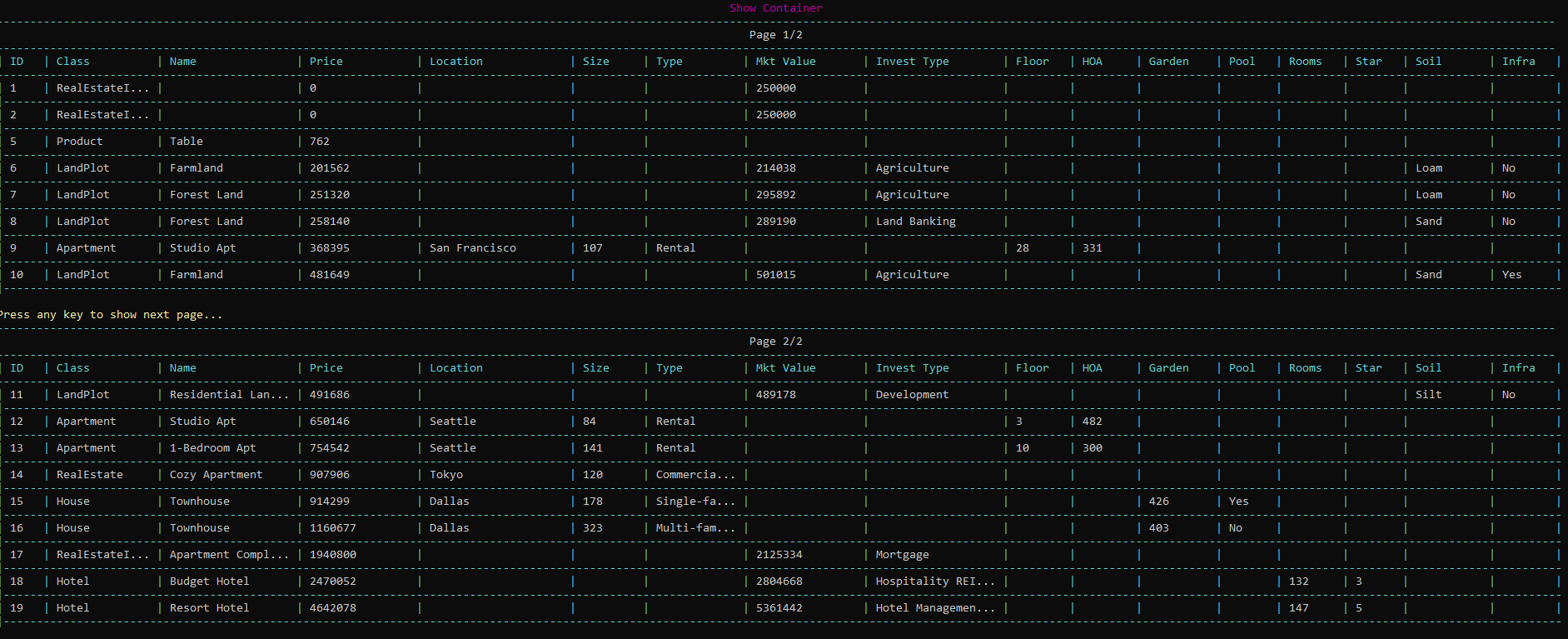


Рис. 4.3. Тестування коду видалення за iндексом

# Робота з індексаторами у класі-контейнері

## Алгоритм коду

Повний код програми відображено у додатку А.

Для реалізації доступу до елементів контейнера за різними критеріями за допомогою синтаксису індексаторів, клас Container було модифіковано наступним чином:

1. Додавання відстеження порядку вставки:

Додано приватний масив insertionOrder типу int[], паралельний до основного масиву items. Додано приватний лічильник nextInsertionId типу int, який інкрементується при кожному додаванні елемента.

Метод Add модифіковано таким чином, щоб при додаванні нового об'єкта в items[count], відповідний унікальний ідентифікатор порядку вставки (nextInsertionId) зберігався у insertionOrder[count]. При розширенні масивів items та insertionOrder розширюються синхронно.

2. Реалізація індексатора за порядковим номером (int):

Створено індексатор public object this[int i]. У get-аксесорі виконується пошук у масиві insertionOrder індексу j, для якого insertionOrder[j] == i (де i – це шуканий порядковий номер вставки).

Якщо такий індекс j знайдено, повертається елемент items[j]. Якщо порядковий номер i не знайдено (або він виходить за межі діапазону), генерується виняток IndexOutOfRangeException.

3. Реалізація індексаторів за ім'ям (string) та ціною (decimal):

Створено індексатори public object[] this[string i] та public object[] this[decimal i]. Вони повертають масив object[], оскільки ім'я або ціна не обов'язково є унікальними.

Для реалізації пошуку за параметром створено універсальний приватний метод GetItemsByParameter<T>(string param, T value). Цей метод ітерує по масиву items (до count), і для кожного елемента за допомогою рефлексії (через інший допоміжний метод GetPropertyValue<T>) отримує значення властивості з ім'ям param.

Якщо отримане значення властивості дорівнює шуканому value, об'єкт додається до результуючого масиву.Індексатори за string та decimal просто викликають GetItemsByParameter, передаючи відповідно "Name" або "Price" як ім'я параметра.

4. Модифікація методів RemoveById та Sort:

Метод RemoveById змінено так, щоб при зсуві елементів у масиві items аналогічний зсув відбувався і в масиві insertionOrder, зберігаючи відповідність між елементом та його порядковим номером вставки відносно інших елементів, що залишилися.

Метод Sort модифіковано для синхронного сортування обох масивів (items та insertionOrder). Коли два елементи в items міняються місцями на основі порівняння їх цін, відповідні їм значення в insertionOrder також міняються місцями.

5. Оновлення ToString() в ієрархії класів:

Методи ToString() у похідних класах (RealEstate, Apartment тощо) були оновлені для включення виклику base.ToString(), що дозволяє виводити інформацію з базових класів, роблячи вивід більш повним та структурованим.

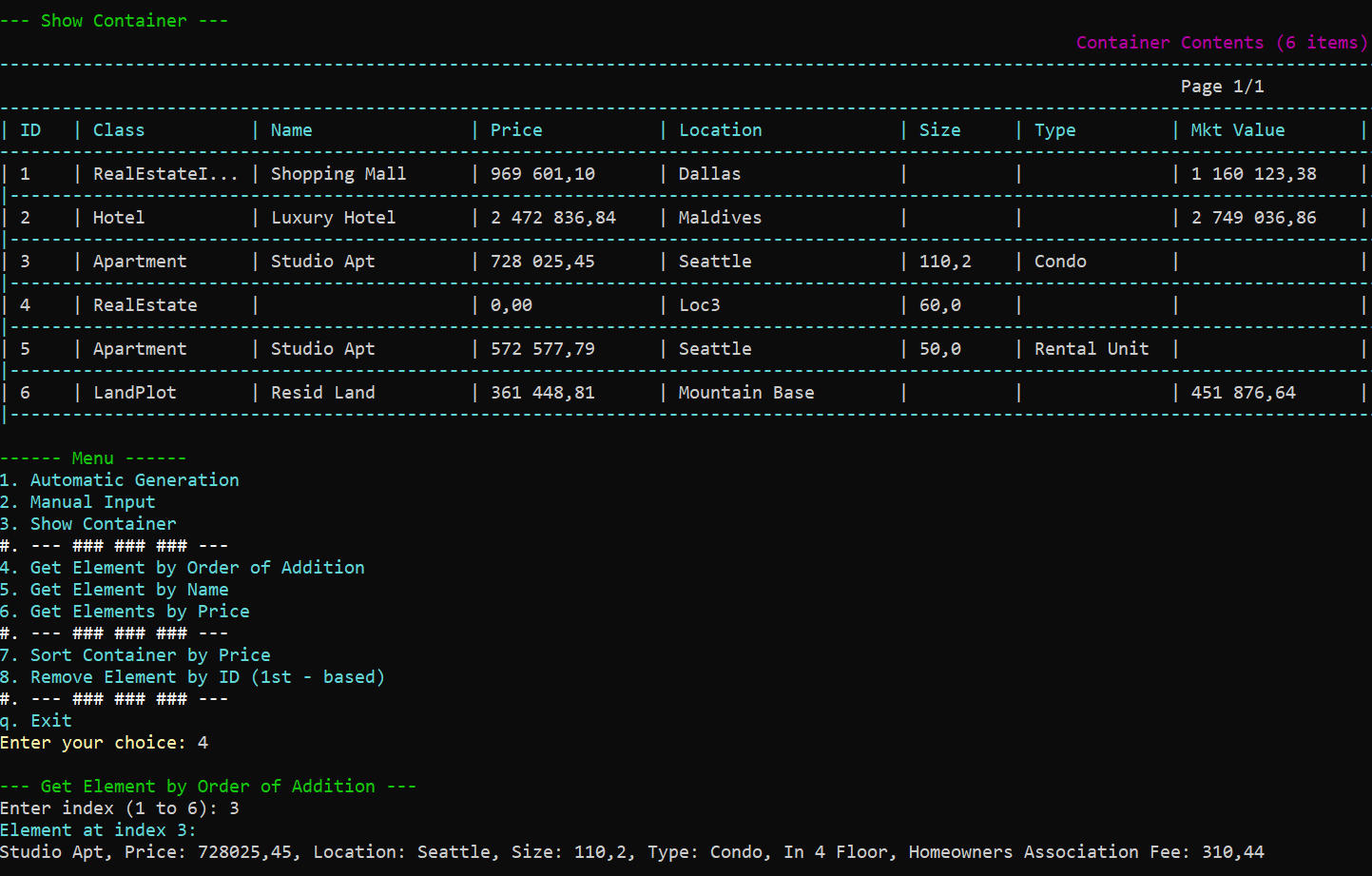
6. Оновлення консольної програми тестування:

Головне меню програми доповнено пунктами для тестування нових індексаторів (отримання за порядковим номером, за ім'ям, за ціною).Реалізовано відповідні функції (GetElementByOrderOfAddition, GetElementByName, GetElementsByPrice), які запитують у користувача необхідний ключ (індекс, ім'я або ціну) та виводять результат роботи індексатора.

Додано метод DemonstrateIndexers для автоматичної демонстрації роботи індексаторів після генерації випадкових даних.Оновлено обробку винятків для коректного перехоплення IndexOutOfRangeException, що може генеруватися індексаторами.

## Тестування коду

Тестування індексатора за порядковим номером зображено на рис. 5.1.



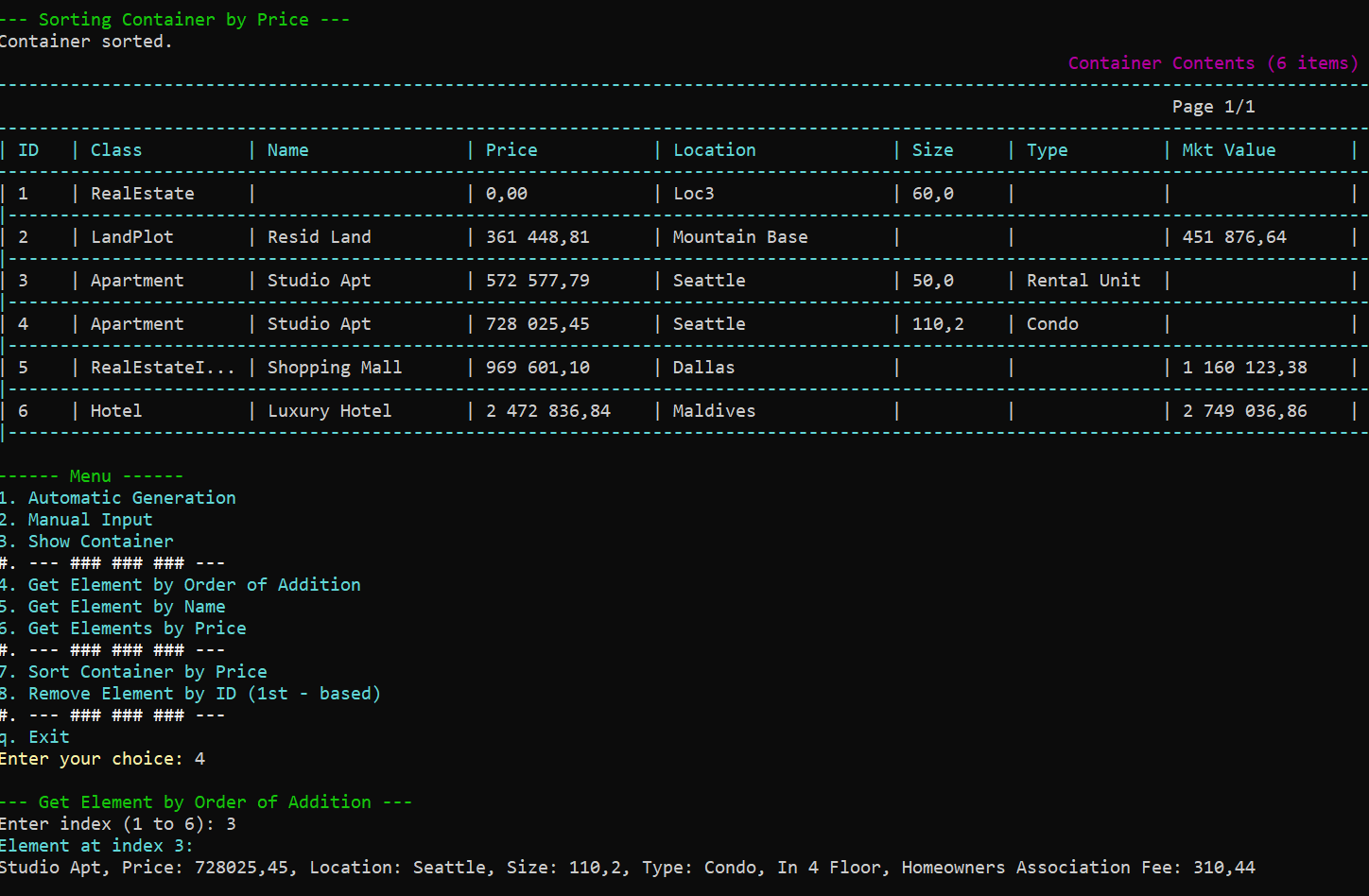


Рис. 5.1. Тестування індексатора за ім'ям

Тестування індексатора за ім'ям відображено на рис. 5.2.

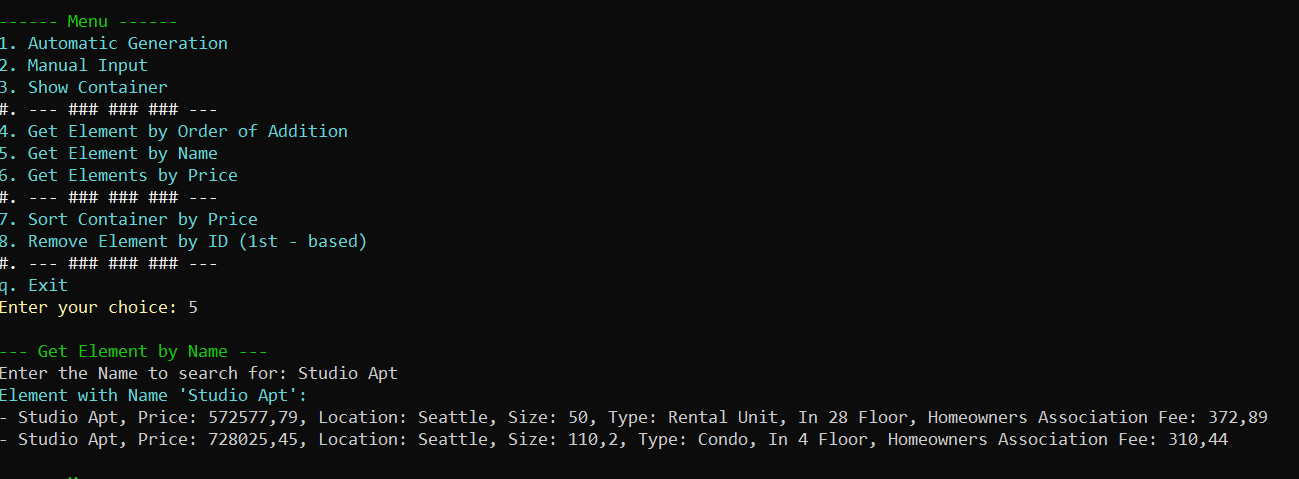


Рис. 5.2. Тестування індексатора за ім'ям

Тестування індексатора за ціною зображено на рис. 5.3.

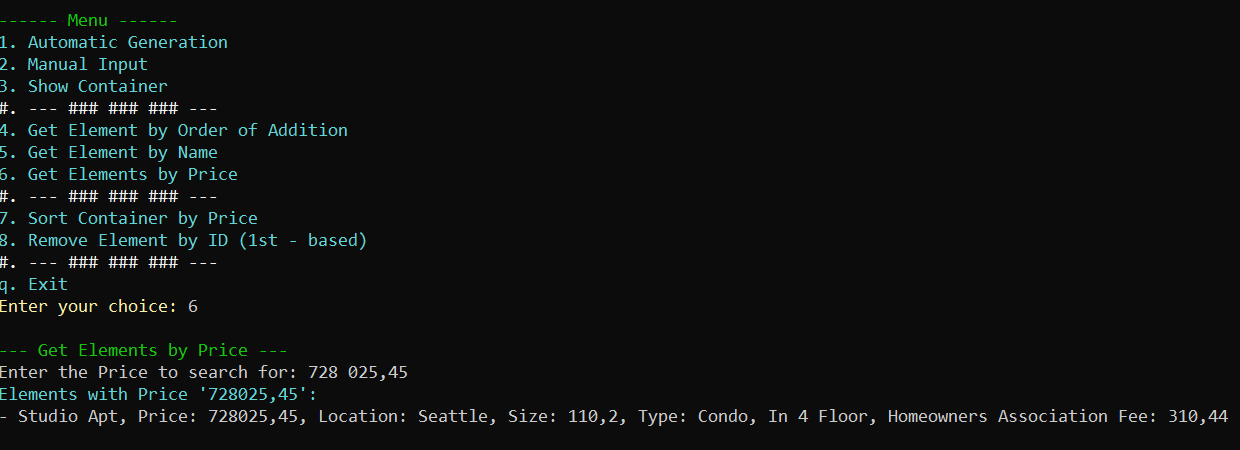


Рис. 5.3. Тестування індексатора за ціною

# Робота з опрацюванням виняткових ситуацій у .NET Framework

## Алгоритм коду

Повний код програми відображено у додатку А.

1. Створення власного класу винятку ValueLessThanZero

Для обробки виняткових ситуацій, пов'язаних з некоректними значеннями властивостей, було створено клас ValueLessThanZero, що є спадкоємцем від стандартного класу Exception. Цей клас дозволяє більш конкретно ідентифікувати помилки, пов'язані зі спробами присвоєння властивостям об'єктів від'ємних або нульових значень, де це є неприпустимим за логікою програми.

2. Модифікація конструкторів класів ієрархії товарів

У конструктори класів Product, RealEstate, RealEstateInvestment, Apartment, House, Hotel, LandPlot було додано перевірки вхідних параметрів на валідність. Зокрема, перевіряється, щоб ціна, розмір, ринкова вартість, кількість кімнат, рейтинг готелю та інші числові значення були більші за нуль (або не від’ємні, залежно від контексту). У разі виявлення некоректного значення, конструктор генерує виняток типу ValueLessThanZero, вказуючи назву властивості, що спричинила помилку.

3. Обробка винятків у консольній програмі

У методах рандомної та ручної головної програми реалізовано блоки try-catch для обробки можливих винятків типу ValueLessThanZero та FormatException (для випадків некоректного введення даних користувачем).

Блок try охоплює код створення об'єктів класів ієрархії товарів.

Блок catch (ValueLessThanZero ex) перехоплює винятки типу ValueLessThanZero, що генеруються конструкторами класів при спробі створення об'єктів з некоректними даними. У цьому блоці виводиться повідомлення про помилку, використовуючи властивість Message винятку, червоним кольором для наочності.

Блок catch (FormatException) обробляє винятки, що виникають при спробі перетворення введених користувачем рядків у числові типи даних у режимі ручного введення.

Блок finally забезпечує скидання кольору консолі до стандартного після обробки винятку або успішного виконання коду.

4. Обробка винятків у класі Container:

Методи класу Container, що можуть призвести до помилок через некоректні індекси, вже генерували стандартний виняток IndexOutOfRangeException. У метод Sort додано загальний блок try-catch (Exception e) для перехоплення будь-яких можливих помилок під час сортування (наприклад, при невдалій спробі отримати властивість Price через рефлексію).

## Тестування коду

Тестування коду обробки винятку ValueLessThanZero відображено на рис. 6.1.

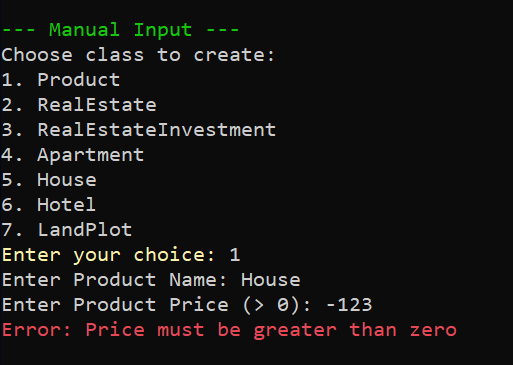


Рис. 6.1. Тестування коду обробки винятку ValueLessThanZero

Тестування коду обробки IndexOutOfRangeException відображено на рис. 6.2.

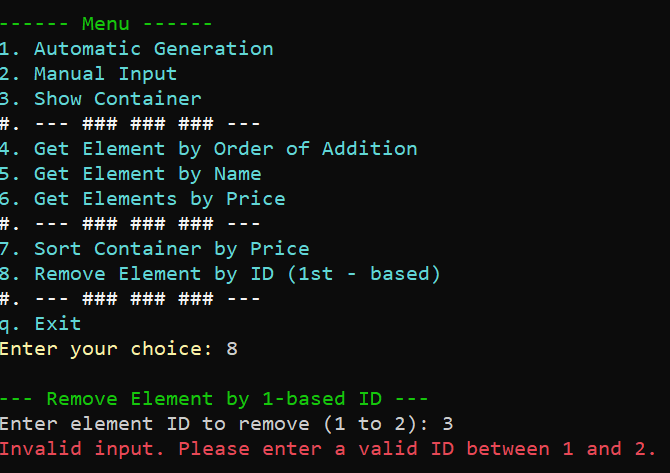


Рис. 6.2. Тестування коду обробки IndexOutOfRangeException

# Робота з інтерфейсами у .NET Framework

## Алгоритм коду

Повний код програми відображено у додатку А.

1. Створення інтерфейсу IName

Створено файл IName.cs, де оголошено публічний інтерфейс IName.Вказано, що IName успадковується від стандартного інтерфейсу System.IComparable.В інтерфейсі оголошено сигнатуру властивості Name типу string з методами доступу get та set.

2. Реалізація IName у класі Product

Клас Product (базовий для всієї ієрархії) модифіковано для реалізації інтерфейсу IName. Оскільки клас Product вже мав публічну властивість string Name { get; set; }, це задовольнило вимогу інтерфейсу щодо властивості Name.

Реалізовано метод CompareTo(object obj), який вимагається інтерфейсом IComparable (успадкованим через IName). Метод порівнює поточний об'єкт з іншим об'єктом obj за значенням їх властивостей Name, використовуючи StringComparer.OrdinalIgnoreCase для порівняння без урахування регістру. Це дозволяє сортувати об'єкти за ім'ям.

3. Модифікація класу Container для роботи з IName

Тип внутрішнього масиву items змінено з object[] на IName?[]. Параметр методу Add змінено на тип IName.Типи повернення методів GetItems, GetItemsByParameter, GetInstanceByInsertionId та індексаторів оновлено до IName або IName[] відповідно. Це забезпечує типізацію контейнера – він може зберігати лише об'єкти, що реалізують IName.

4. Оновлення індексаторів у Container

Індексатор за порядковим номером (public IName? this[int id]) залишено. Він повертає об'єкт типу IName? за його унікальним ID вставки. Додано set-аксесор для можливості заміни об'єкта за його ID вставки.

Індексатор за ім'ям (public IName[] this[string i]) залишено. Він використовує метод GetItemsByParameter для пошуку всіх об'єктів типу IName з відповідним ім'ям і повертає масив IName[].

Індексатор за ціною (public IName[] this[decimal i]) було видалено (закоментовано), відповідно до завдання.

5. Модифікація консольної програми

Оновлено меню програми: видалено пункт для отримання елементів за ціною (пункт 6 у коді закоментовано).

Додано нові пункти меню (6 та 7) для тестування функціоналу зміни елементів за порядковим номером або за ім'ям (використовуючи set-аксесор індексатора та рефлексію у методі ModifyProperty).

Функції взаємодії з користувачем (GetElementBy..., ChangeItemBy..., ManualInput тощо) оновлено для коректної роботи з об'єктами типу IName.

У методі AutomaticGeneration та методах ручного створення об'єктів тепер створюються та додаються об'єкти, що гарантовано реалізують IName.

## Тестування коду

Тестування додавання згенерованих та введених вручну об'єктів з інтерфейсом IName відображено на рис. 7.1 та рис. 7.2, відображення контейнера на рис. 7.3.

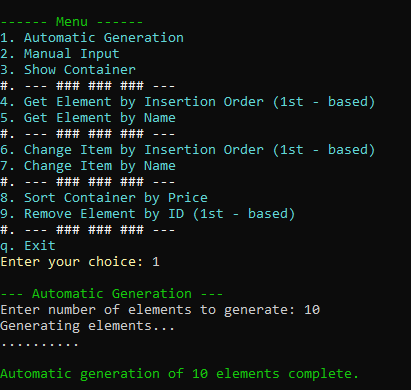


Рис. 7.1. Тестування додавання згенерованих об'єктів

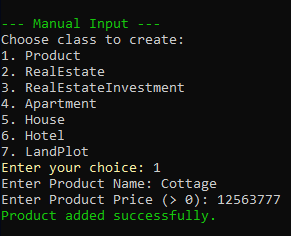


Рис. 7.2. Тестування введених вручну об'єкту

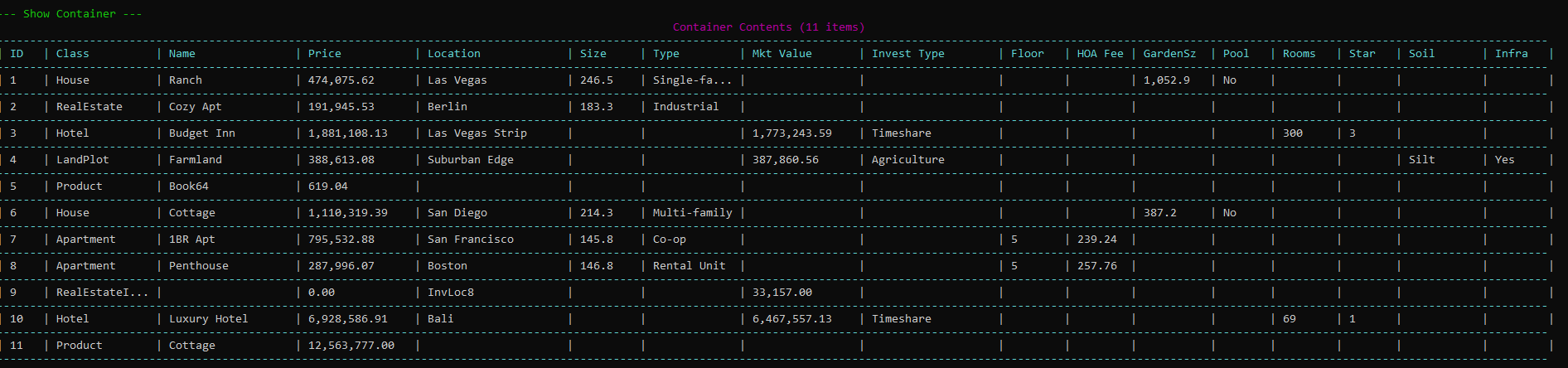


Рис. 7.3. Відображення контейнера

# Робота з узагальненнями та реалізація різних типів контейнерів

## Алгоритм коду

Повний код програми відображено у додатку А.

1. Створення узагальненого інтерфейсу IName<T>

У файлі IName.cs додано визначення інтерфейсу IName<T>. Цей інтерфейс містить лише одну вимогу – наявність властивості string Name { get; set; }. Він не успадковується від IComparable.

2. Реалізація IName<T> у класах ієрархії

Кожен клас ієрархії (Product, RealEstate, Apartment тощо) було модифіковано для реалізації інтерфейсу IName<T>, де T – це сам тип класу. Наприклад, class Product : IName, IName<Product>. Оскільки властивість Name вже існувала і була реалізована для IName, додаткового коду для реалізації властивості з IName<T> не знадобилося.

3. Узагальнення контейнера на основі масиву (Container<T>)

Клас Container перетворено на узагальнений клас Container<T>. Додано обмеження where T : class, IName, що гарантує: T має бути посилальним типом (класом) і реалізовувати інтерфейс IName (а отже, мати властивість Name та реалізацію IComparable).

Внутрішній масив items тепер має тип T?[]. Методи (Add, RemoveById, GetItems, GetItemsByParameter, індексатори) оновлено для роботи з типом T замість IName чи object. Це забезпечує типобезпеку на етапі компіляції.

Додано допоміжний клас Helper з методом GetPropertyValue для уніфікованого отримання значень властивостей через рефлексію.

4. Створення узагальненого контейнера-списку ContainerLinkedList<T>

Створено новий клас ContainerLinkedList<T> з аналогічним обмеженням where T : class, IName.Внутрішня структура реалізована за допомогою приватного вкладеного класу Node<V>, який зберігає дані типу V та посилання на наступний (Next) і попередній (Previous) вузли. Реалізовано основні операції двозв'язного списку: AddFirst, AddLast, RemoveByIndex (видалення вузла за його поточним індексом у списку), Clear.

Реалізовано властивості First, Last, Count.Збережено логіку відстеження порядку вставки (NextInsertionId, List<int> InsertionOrder).Реалізовано метод Sort, який використовує сортування бінарними вставками (BinaryInsertionSort) для внутрішнього списку List<T>, отриманого з вузлів, і потім оновлює дані у вузлах. Сортування за замовчуванням відбувається за властивістю "Price". Реалізовано індексатори за ID вставки (this[int index]) та за ім'ям (this[string name]), які працюють з вузлами списку та списком ID вставки.

5. Модифікація консольної програми (Program.cs)

Додано enum ContainerType для розрізнення типів контейнерів.Оголошено дві змінні для зберігання контейнерів: containerArray типу Container<IName>? та containerList типу ContainerLinkedList<IName>?. Змінна activeContainerType відстежує, який контейнер зараз активний.

Реалізовано логіку вибору контейнера (AskContainerType) та перемикання між ними (HandleContainerSelectionAndAction). При перемиканні попередній контейнер очищується (стає null). Основний цикл Main та всі методи-обробники (Handle...) тепер перевіряють activeContainerType і викликають відповідні методи активного контейнера (containerArray або containerList).

Додано допоміжні функції для роботи з обома типами контейнерів: IsContainerEmpty, GetActiveContainerCount, FindIndexByReference, GetInsertionIdForItem, GetItemByCurrentIndex тощо. Оновлено методи виведення таблиці для роботи з обома типами контейнерів.Оновлено методи автоматичної генерації та ручного введення для додавання елементів до активного контейнера.

## Тестування коду

Тестування узагальнення перевірено на етапі компіляції, що контейнери можуть приймати лише об'єкти класів, що реалізують IName зображено на Рис. 8.1 та Рис. 8.2, відображення контейнера на Рис. 8.3.

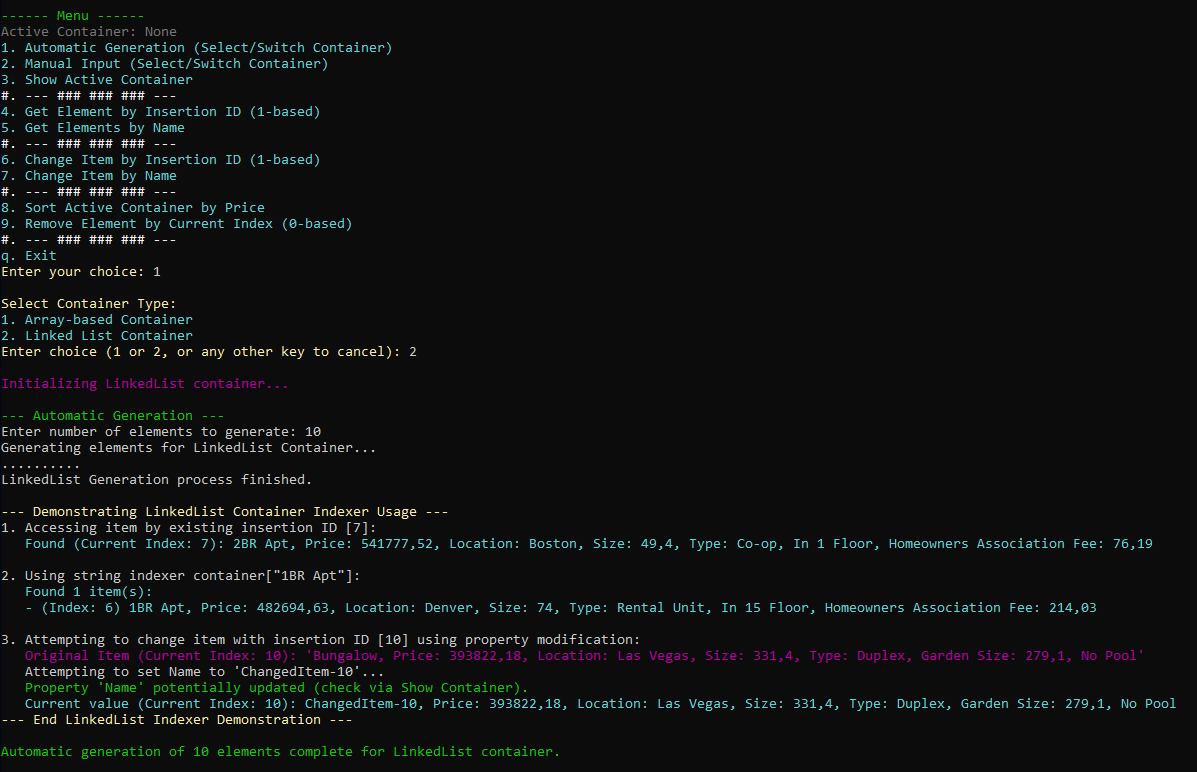


Рис. 8.1. Тестування додавання згенерованих об'єктів

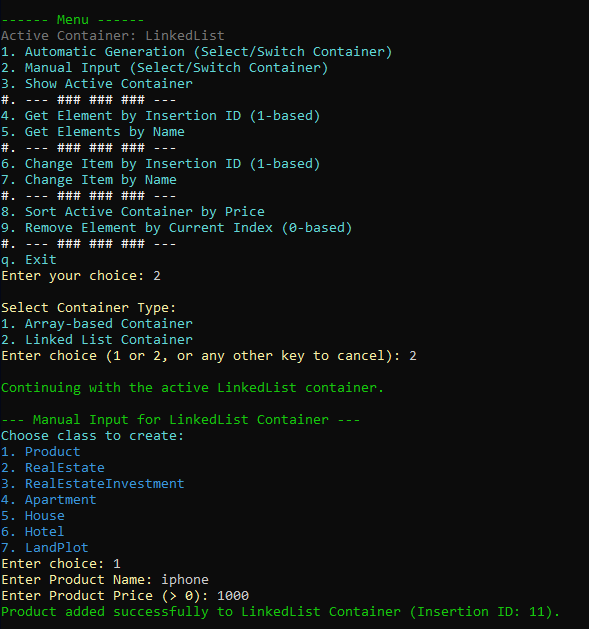


Рис. 8.2. Тестування введених вручну об'єкту

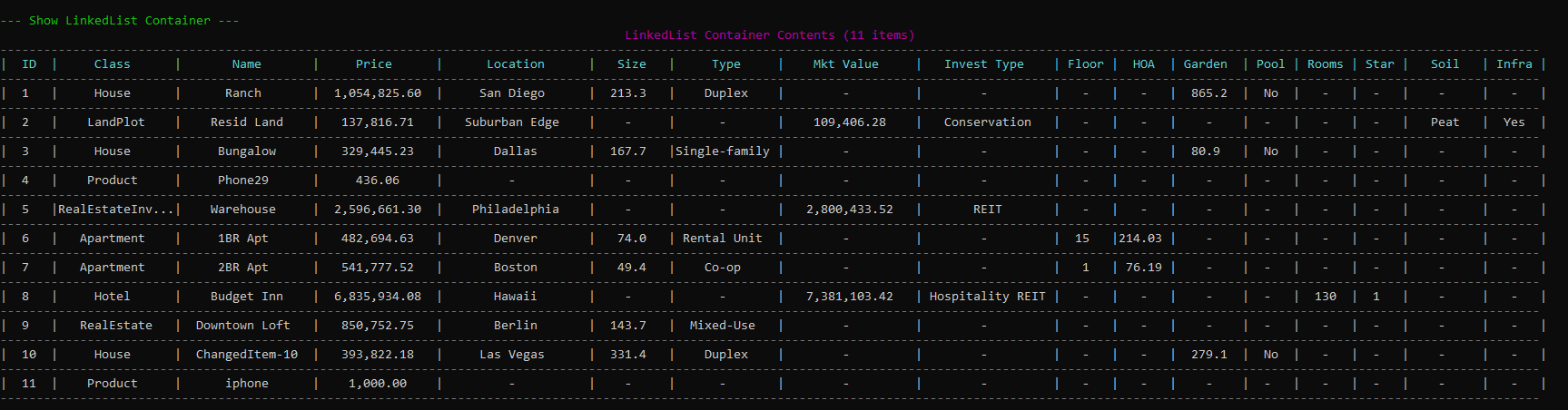


Рис. 8.3. Відображення контейнера

# Реалізація ітераторів у контейнерах у .NET Framework

## Алгоритм коду

Повний код програми відображено у додатку А.

Для забезпечення можливості ітерації по контейнерах за допомогою foreach без використання yield, було реалізовано інтерфейси IEnumerable<T> та IEnumerator<T> наступним чином:

1. Реалізація ітератора для Container<T> (на основі масиву)

Клас Container<T> тепер реалізує інтерфейс IEnumerable<T>. Було видалено попередню реалізацію методу GetEnumerator(), яка використовувала yield. Створено внутрішній (private nested) клас ContainerEnumerator, який реалізує інтерфейс IEnumerator<T>. Це необхідно, оскільки сам контейнер на основі масиву не може легко відстежувати поточну позицію ітерації для кількох одночасних циклів foreach без зовнішнього об'єкта-ітератора.

Клас ContainerEnumerator містить:

* Посилання на екземпляр Container<T>, щоб мати доступ до масиву items та лічильника count.
* Приватне поле \_currentIndex (int), яке зберігає поточну позицію в масиві під час ітерації (ініціалізується значенням -1).
* Реалізацію властивості Current (з IEnumerator<T>): повертає елемент items[\_currentIndex].
* Реалізацію методу MoveNext(): інкрементує \_currentIndex і повертає true, якщо нова позиція знаходиться в межах дійсних елементів контейнера (менше count), інакше повертає false. Важливо: Цей метод може пропускати null елементи, якщо вони потенційно можуть з'явитися в масиві між дійсними елементами (хоча при поточній реалізації RemoveById це малоймовірно).
* Реалізацію методу Reset(): скидає \_currentIndex на -1.
* Реалізацію методу Dispose(): порожній метод, оскільки ітератор не керує зовнішніми ресурсами.

Метод GetEnumerator() класу Container<T> тепер створює та повертає новий екземпляр ContainerEnumerator(this).Явно реалізовано також не узагальнений метод IEnumerable.GetEnumerator(), який викликає узагальнений GetEnumerator().

2. Реалізація ітератора для ContainerLinkedList<T>

Цей клас вже реалізував інтерфейси IEnumerable<T> та IEnumerator<T> у попередній роботі, причому сам клас виступав у ролі свого ітератора. Цей підхід не використовує yield і відповідає вимогам завдання.

Клас містить поле \_currentNode, яке зберігає посилання на поточний вузол під час ітерації.Властивість Current повертає дані (Data) з \_currentNode.

Метод MoveNext() переміщує \_currentNode на наступний вузол (\_currentNode.Next) і повертає true, якщо наступний вузол існує. При першому виклику він встановлює \_currentNode на голову списку (\_head).

Метод Reset() скидає \_currentNode на null, готуючи до нової ітерації.Метод GetEnumerator() просто повертає this, оскільки клас сам є ітератором.

## Тестування коду

Тестування узагальнення перевірено на етапі компіляції, що контейнери можуть приймати лише об'єкти класів, що реалізують IName зображено на Рис. 9.1 та Рис. 9.2, відображення контейнера на Рис. 9.3.

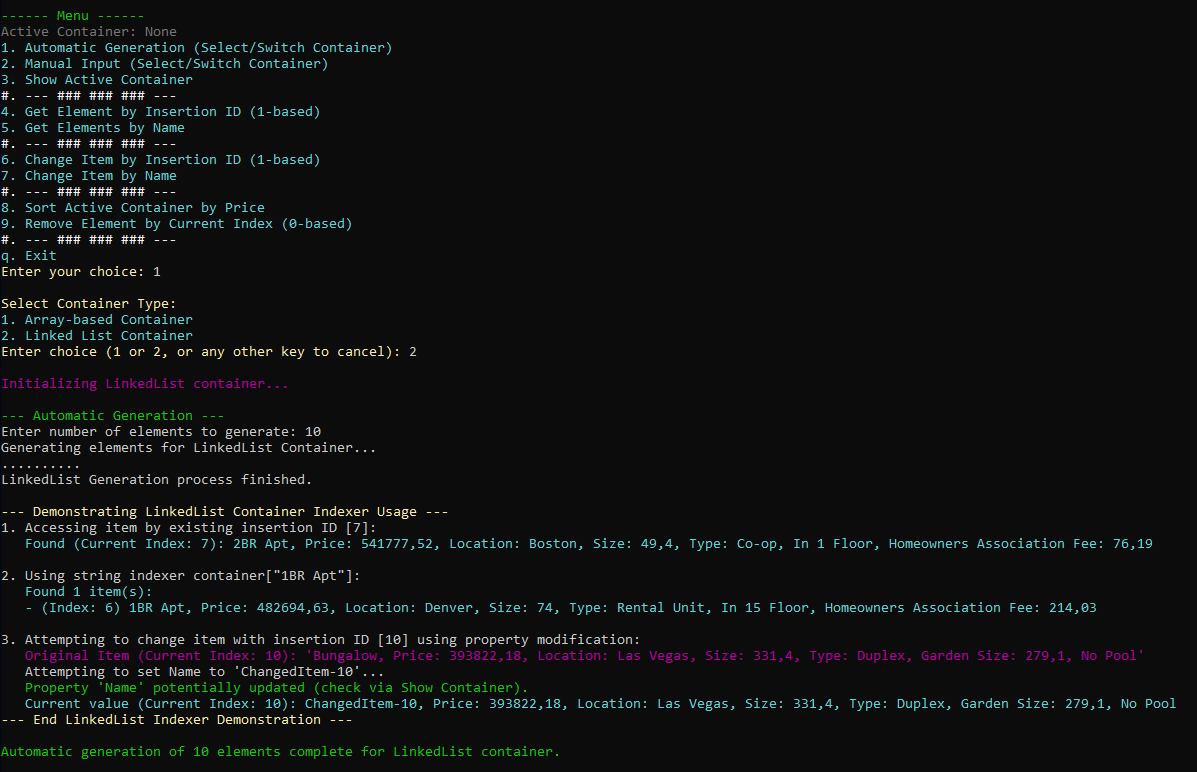


Рис. 9.1. Тестування додавання згенерованих об'єктів

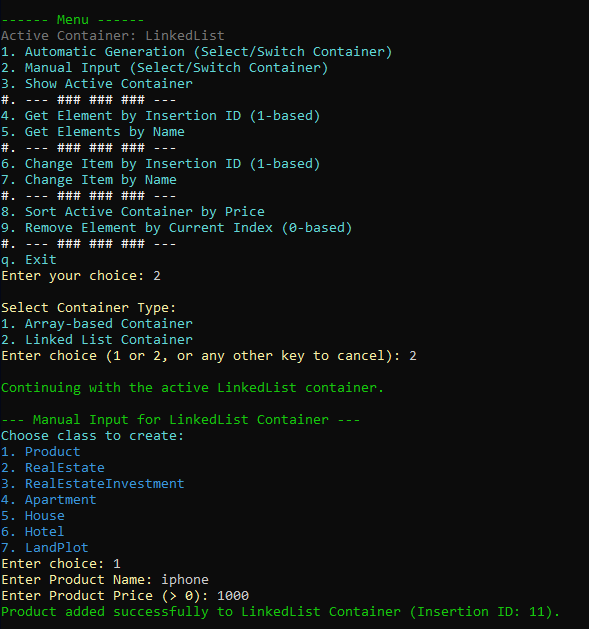


Рис. 9.2. Тестування введених вручну об'єкту

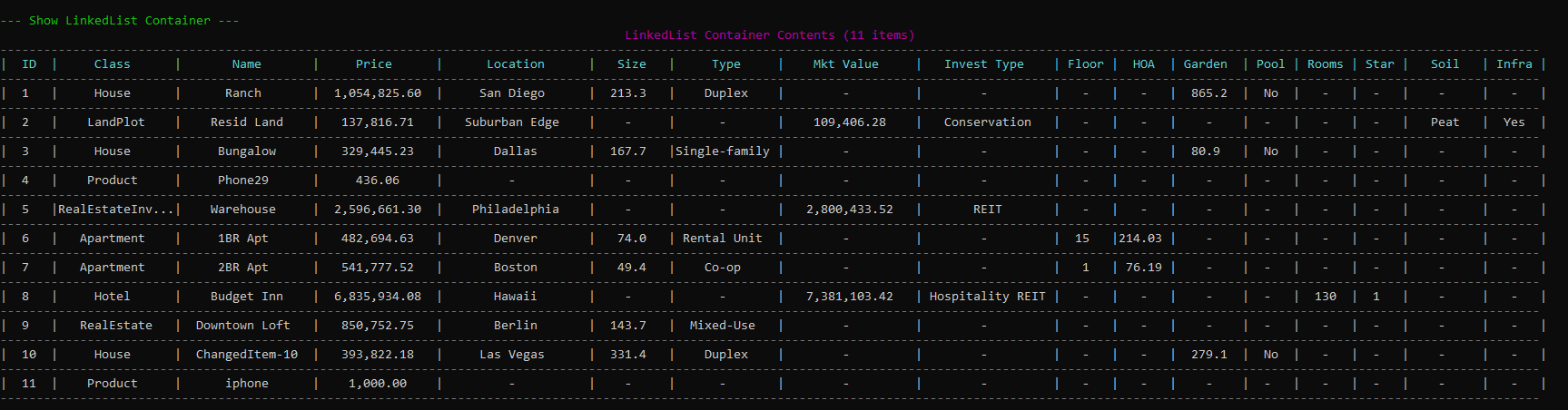


Рис. 9.3. Відображення контейнера

# Робота з іменованими ітераторами (генераторами)

## Алгоритм коду

Повний код програми відображено у додатку А.

Для реалізації завдання було додано нові методи-генератори до класів Container<T> та ContainerLinkedList<T>, а також оновлено консольну програму.

1. Реалізація генераторів для Container<T> (масив)

Зворотний порядок (GetReverseArray()):

* Створюється копія внутрішнього масиву items за допомогою items.Clone().
* Використовується стандартний метод Array.Reverse() для зміни порядку елементів у копії масиву.
* За допомогою циклу foreach та yield return повертаються ненульові елементи з реверсованої копії масиву. Оригінальний масив items залишається незмінним.

За підрядком у імені (GetArrayWithSublineInName(string subline)):

* Метод приймає рядок subline як параметр.
* За допомогою циклу foreach перебираються елементи оригінального масиву items.
* Для кожного ненульового елемента перевіряється, чи його властивість Name містить (Contains()) заданий subline.
* Якщо умова виконується, елемент повертається за допомогою yield return.

Сортування за ціною (GetSortedByArrayPrice()):

* Створюється копія масиву items.Реалізовано сортування бульбашкою для копії масиву, використовуючи метод CompareByPrice() для порівняння.
* За допомогою foreach та yield return повертаються елементи з відсортованої копії.

Сортування за ім'ям (GetSortedArrayByName()):

* Аналогічно до сортування за ціною, але використовується метод CompareTo() (який порівнює за ім'ям) для сортування копії масиву.
* За допомогою foreach та yield return повертаються елементи з відсортованої копії.

## Тестування коду

Тестування генераторiв за: зворотним порядоком, за підрядком у імені, сортування за ціною, сортування за ім'ям - зображено на Рис. 10.1 та Рис. 10.2, Рис. 10.3, Рис. 10.4.

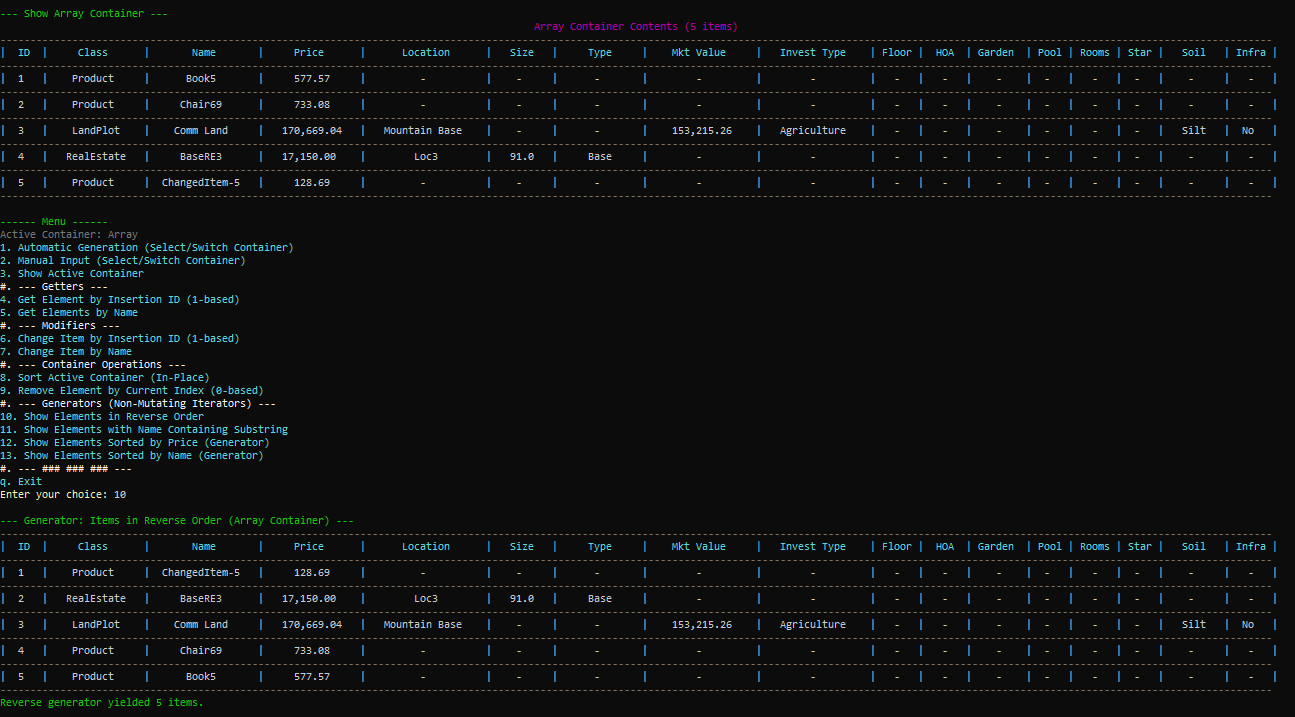


Рис. 10.1. Тестування за зворотним порядоком

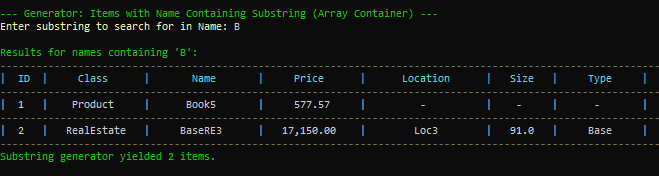


Рис. 10.2. Тестування за підрядком у імені

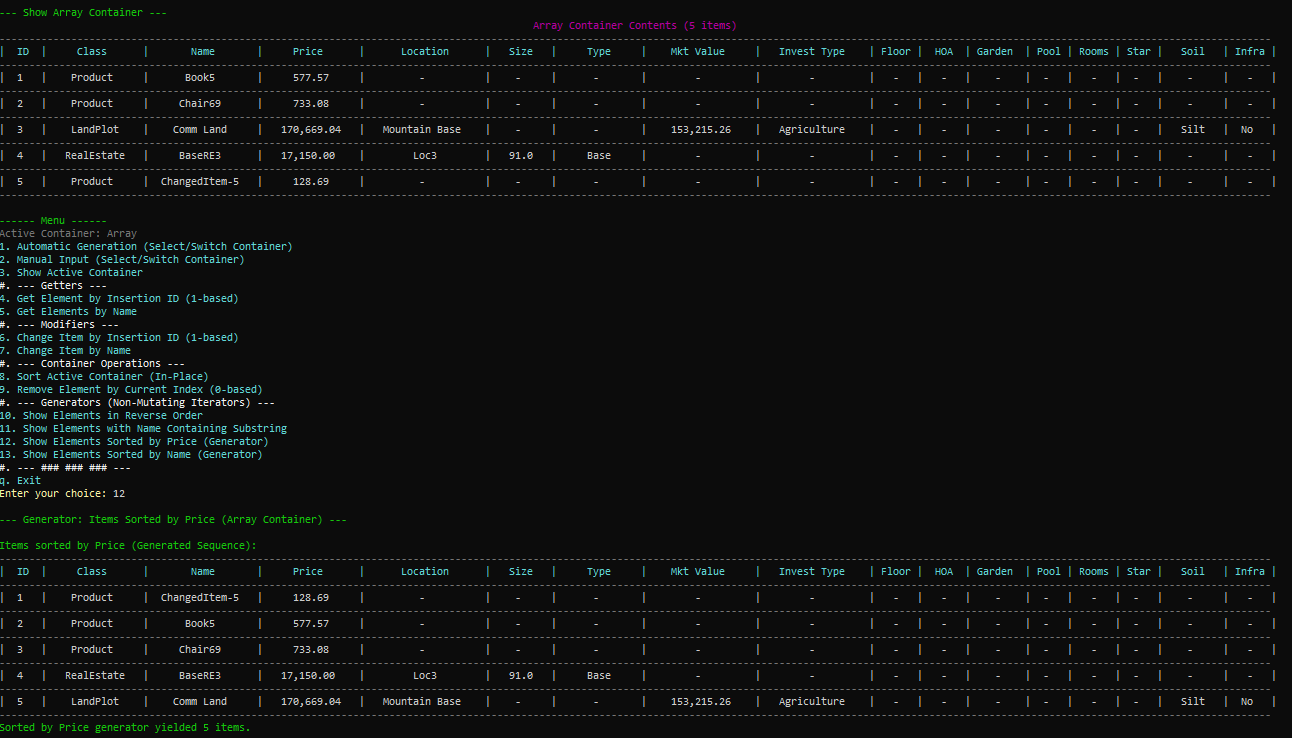


Рис. 10.3. Відображення сортування за ціною

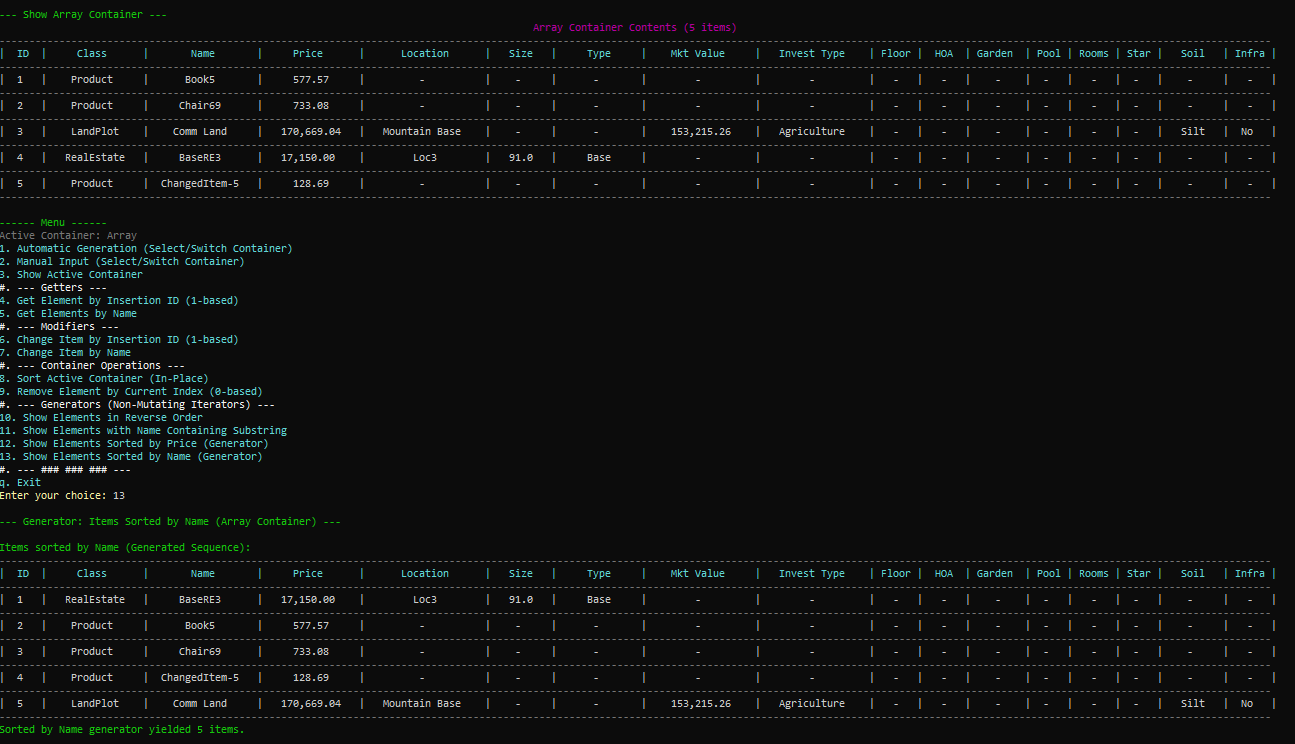


Рис. 10.4. Відображення сортування за ім'ям

# Реалізація власної бінарної серіалізації та десеріалізації контейнерів

## Алгоритм коду

Повний код програми відображено у додатку А.

Для реалізації власного механізму бінарної серіалізації/десеріалізації було вжито наступних заходів:

Визначення інтерфейсу ICustomSerializable:

* Створено інтерфейс ICustomSerializable, який успадковується від IName.
* Цей інтерфейс додає одну вимогу: наявність методу void Serialize(BinaryWriter writer), який відповідатиме за запис стану об'єкта у бінарний потік.
* Хоча інтерфейс не може вимагати статичних методів, було прийнято конвенцію, що кожен клас, який реалізує ICustomSerializable, також надаватиме статичний метод Deserialize(BinaryReader reader) для відновлення об'єкта з потоку.

Реалізація ICustomSerializable у класах ієрархії:

* Кожен клас ієрархії товарів (Product, RealEstate, Apartment тощо) було модифіковано для реалізації інтерфейсу ICustomSerializable.
* Метод Serialize(BinaryWriter writer): У кожному класі цей метод записує значення всіх власних та успадкованих публічних властивостей у переданий BinaryWriter у чітко визначеному порядку. Використовуються відповідні методи writer.Write() для різних типів даних (string, decimal, int, double, bool).
* Статичний метод Deserialize(BinaryReader reader): У кожному класі цей метод зчитує дані з BinaryReader у тому ж порядку, в якому вони були записані методом Serialize. Створюється новий екземпляр класу (new ClassName()) і прочитані значення присвоюються його властивостям. Метод повертає створений та заповнений об'єкт.

Створено статичний клас ContainerSerializer для інкапсуляції логіки серіалізації та десеріалізації контейнерів.

* Метод SerializeContainer(object container, string name):
  + Приймає об'єкт контейнера та ім'я файлу.
  + Визначає тип контейнера (Container<IName> чи ContainerLinkedList<IName>).
  + Створює FileStream та BinaryWriter.
  + Записує метадані:
    - Назву типу контейнера (без узагальнених параметрів, напр., "Container").
    - Кількість елементів у контейнері.
  + Ітерує по елементах активного контейнера.
  + Для кожного елемента:
    - Перевіряє, чи реалізує він ICustomSerializable.
    - Записує повне ім'я типу елемента (наприклад, "lb\_11.Classes.Apartment").
    - Викликає метод item.Serialize(writer) для запису стану об'єкта.
* Метод DeserializeContainer(string name):Приймає ім'я файлу.Створює FileStream та BinaryReader.Зчитує метадані: назву типу контейнера та кількість елементів.Використовує словник containerFactories для створення порожнього екземпляра контейнера відповідного типу (Container<IName> або ContainerLinkedList<IName>) за прочитаною назвою.У циклі count разів:Зчитує повне ім'я типу елемента.Використовує словник Deserializers, де ключами є імена типів ("Product", "Apartment" тощо), а значеннями – делегати на відповідні статичні методи Deserialize.Викликає знайдений делегат-десеріалізатор, передаючи reader.Додає відновлений об'єкт до створеного контейнера за допомогою методу Add.Повертає заповнений контейнер.

Модифікація консольної програми (Program.cs):

* Додано нові пункти меню (14 та 15) для виклику серіалізації та десеріалізації.
* Реалізовано методи-обробники HandleSerializeContainer та HandleDeserializeContainer.
* HandleSerializeContainer: перевіряє наявність активного контейнера, запитує ім'я файлу, викликає ContainerSerializer.SerializeContainer, обробляє можливі винятки (IOException, InvalidOperationException тощо).
* HandleDeserializeContainer: запитує ім'я файлу, викликає ContainerSerializer.DeserializeContainer, обробляє винятки (FileNotFoundException, IOException, InvalidDataException). Якщо десеріалізація успішна, пропонує замінити активний контейнер відновленим, оновлює змінні containerArray/containerList та activeContainerType.

## Тестування коду

Тестування проводилося для обох типів контейнерів (Container<T> та ContainerLinkedList<T>) за наступним сценарієм:

* Створення та наповнення контейнера (використовуючи автоматичну генерацію або ручне введення) різними типами об'єктів з ієрархії.
* Виклик функції серіалізації (пункт 14 меню), вказавши ім'я файлу (наприклад, "array" або "list"). Перевірка успішного створення відповідного .bin файлу.
* Виклик функції десеріалізації (пункт 15 меню), вказавши ім'я збереженого файлу.
* Перевірка повідомлення про успішну десеріалізацію та кількість відновлених елементів.
* Підтвердження заміни активного контейнера.
* Виведення вмісту відновленого контейнера на екран (пункт 3 меню). Порівняння виведених даних (кількість елементів, їх типи та значення властивостей) з оригінальним станом контейнера перед серіалізацією.
* Тестування роботи з відновленим контейнером (наприклад, сортування, отримання елемента за індексом) для перевірки цілісності об'єктів.
* Тестування обробки помилок: спроба десеріалізації з неіснуючого файлу (FileNotFoundException), спроба серіалізації без активного контейнера.

Тестування зображенно на рисунках 11.1, 11.2, 11.3, 11.4.

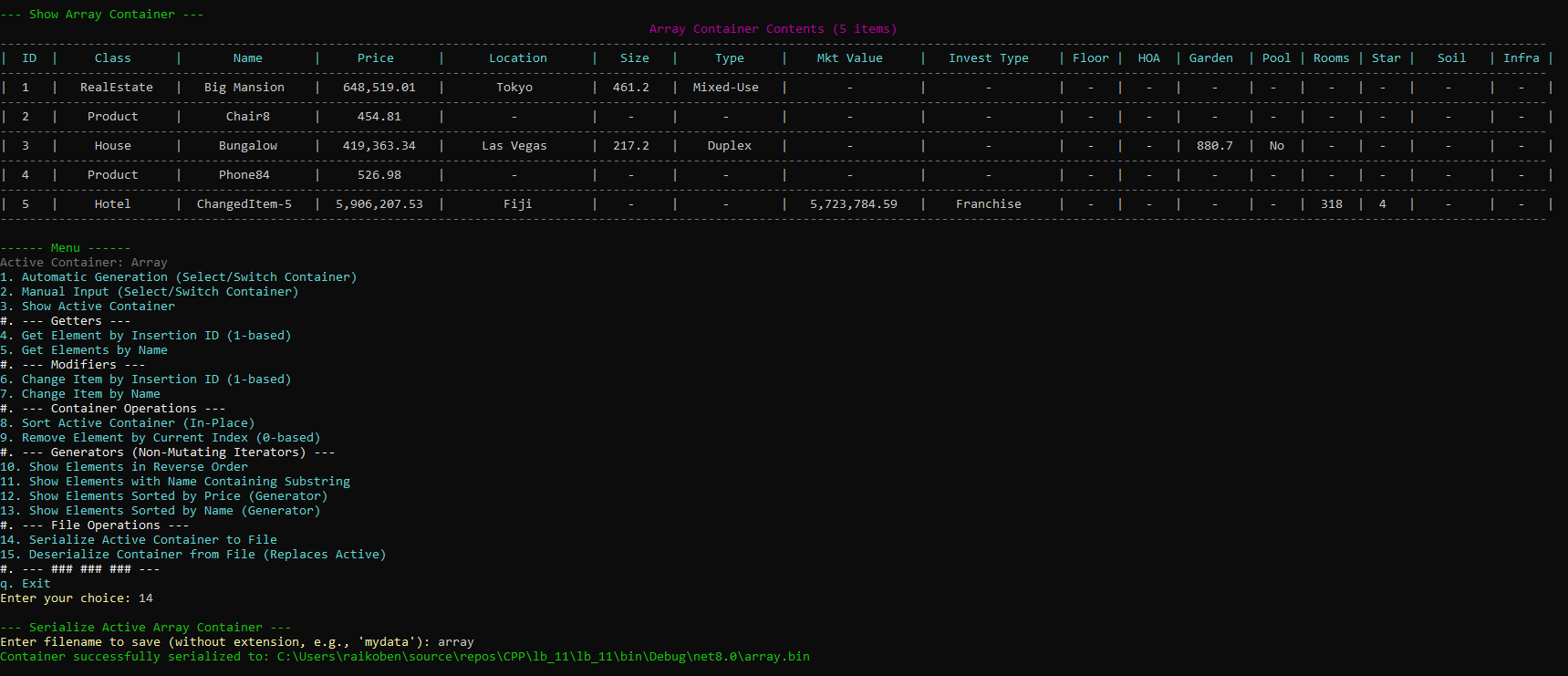


Рис. 11.1. Запис контейнера масиву у файл array.bin

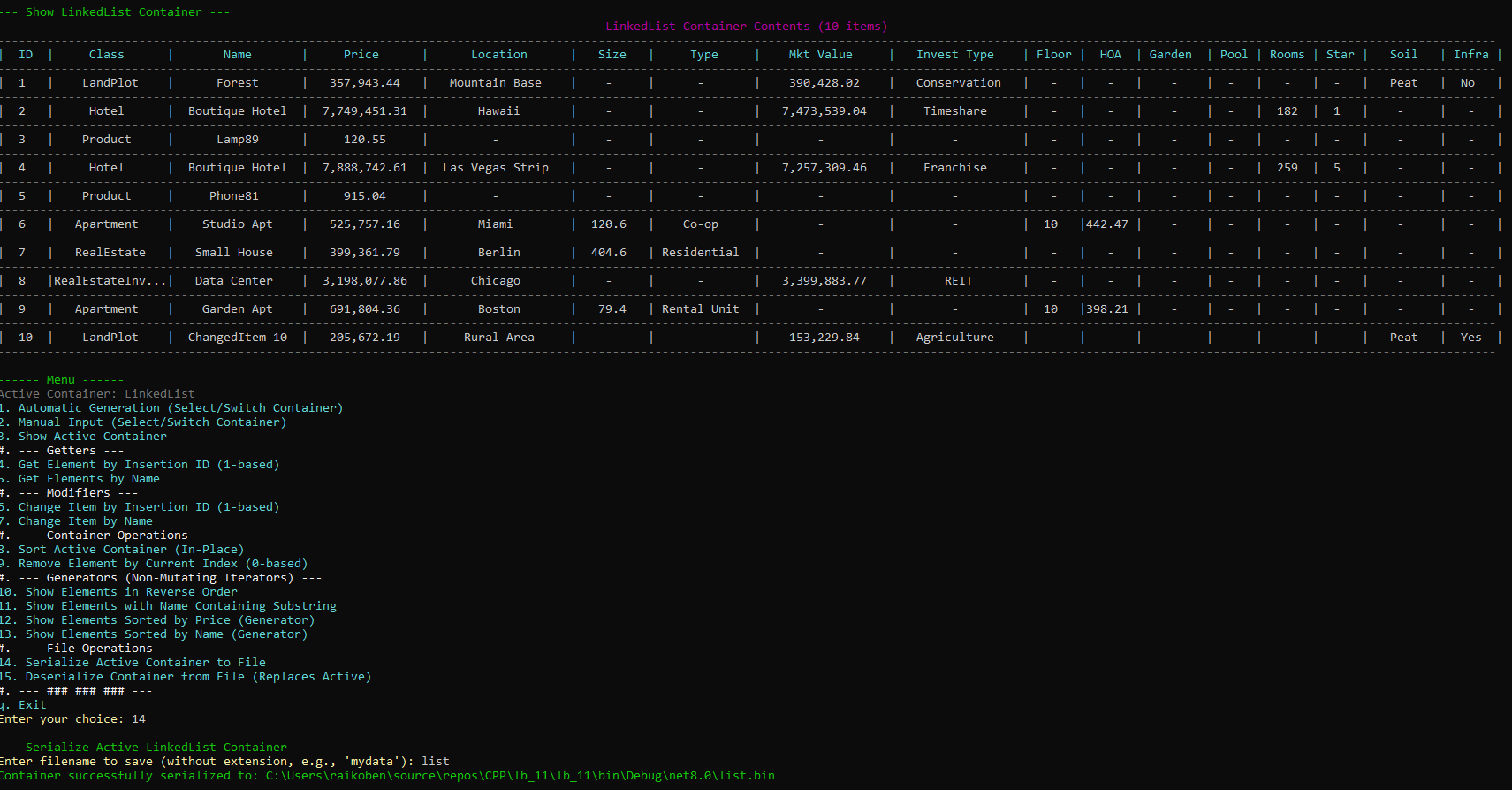


Рис. 11.2. Запис контейнера двозв’язного списку у файл list.bin

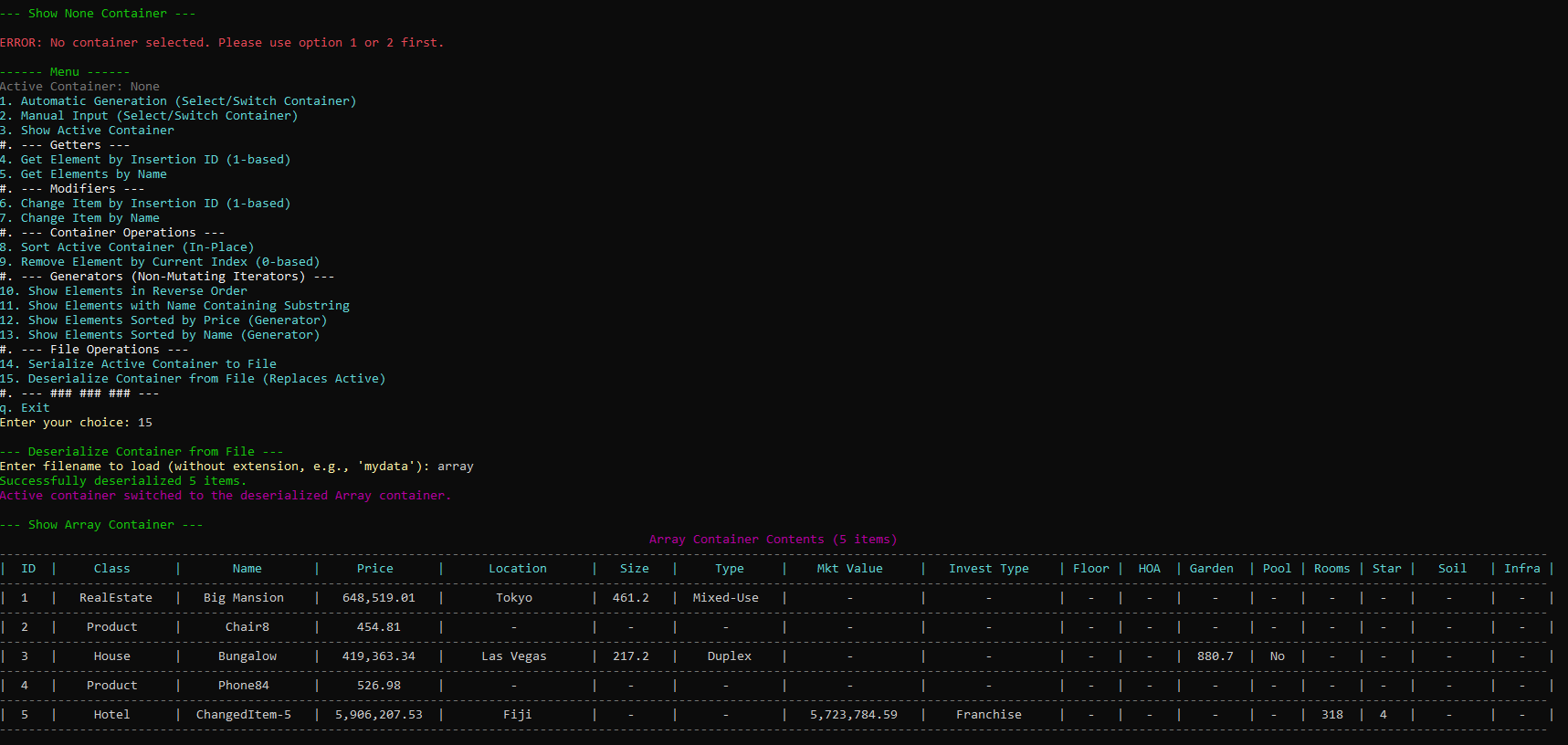


Рис. 11.3. Зчитування та запис у контейнер масив з array.bin

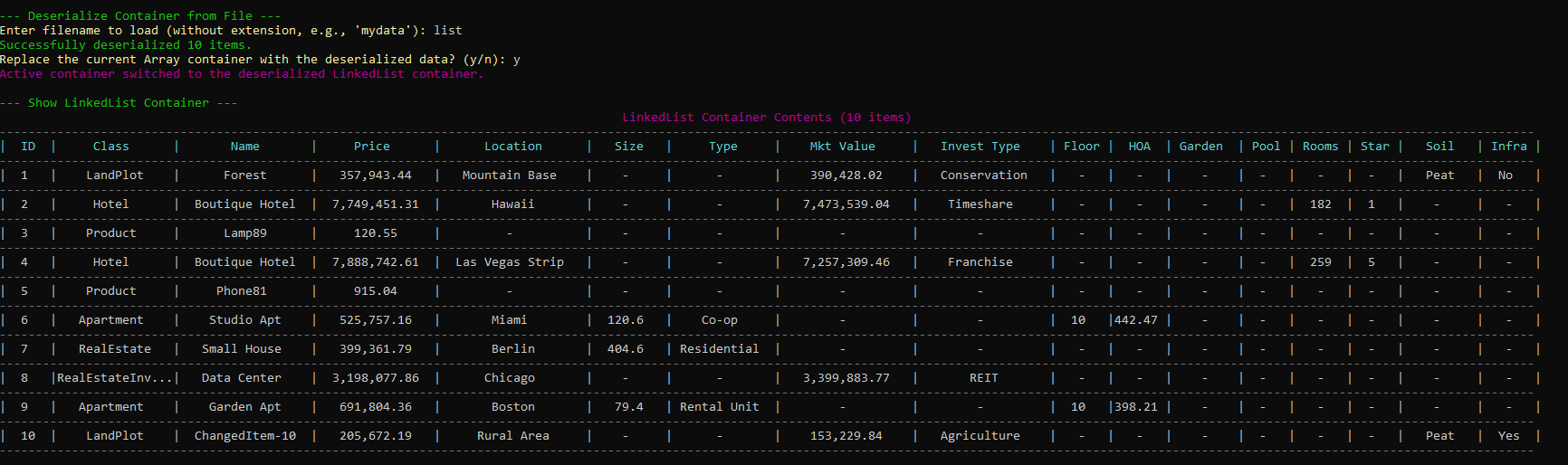


Рис. 11.4. Зчитування та запис у контейнер двозв'язкового списку з list.bin

# Використання делегатів для сортування та пошуку в узагальнених контейнерах

## Алгоритм коду

Повний код програми відображено у додатку А.

Для реалізації поставлених завдань було внесено наступні зміни:Реалізація методу Sort(Comparison<T> comparison):

Для Container<T> (на основі масиву):

* Додано публічний метод void Sort(Comparison<T> comparison).
* Всередині методу викликається стандартний метод Array.Sort(items, 0, count, Comparer<T>. Create(comparison)). Comparer<T>.Create(comparison) створює об'єкт IComparer<T> з переданого делегата Comparison<T>, що дозволяє Array.Sort використовувати власну логіку порівняння. Сортування виконується лише для заповненої частини масиву (від 0 до count).

Для ContainerLinkedList<T> (на основі двозв'язного списку):

* Додано публічний метод void Sort(Comparison<T> comparison).
* Якщо список порожній або comparison дорівнює null, метод завершує роботу.
* Спочатку всі елементи зі списку копіюються у тимчасовий List<T> (за допомогою методу NodeToList()).
* Потім цей List<T> сортується за допомогою стандартного методу List<T>.Sort(comparison).
* Після сортування дані з відсортованого List<T> копіюються назад у вузли двозв'язного списку, зберігаючи їхній зв'язок. (Існуючий механізм сортування BinaryInsertionSort був замінений на більш універсальний підхід з Comparison<T>).

Реалізація методів Find(Predicate<T> match) та FindAll(Predicate<T> match):

Для Container<T> (на основі масиву):

1. Метод Find(Predicate<T> match) використовує LINQ-метод FirstOrDefault() на масиві items (обрізаному до count), передаючи йому предикат match. Він повертає перший елемент, що задовольняє умову, або default(T) (тобто null для класів), якщо такий елемент не знайдено.
2. Метод FindAll(Predicate<T> match) використовує LINQ-метод Where() на масиві items, передаючи предикат match, і повертає IEnumerable<T>, що містить усі знайдені елементи.

Для ContainerLinkedList<T> (на основі двозв'язного списку):

1. Метод Find(Predicate<T> match) ітерує по вузлах списку. Для даних кожного вузла викликається предикат match. Якщо умова виконується, дані вузла повертаються. Якщо жоден елемент не задовольняє умову, повертається default(T).
2. Метод FindAll(Predicate<T> match) ітерує по вузлах списку. Якщо дані вузла задовольняють предикат match, вони додаються до тимчасового List<T>. Наприкінці повертається цей список. (У наданому коді методи Find та FindAll використовують LINQ FirstOrDefault та Where на IEnumerable<T>, що забезпечується реалізацією GetEnumerator класом ContainerLinkedList<T>.)

Визначення делегатів Comparison<IName> у Program.cs:

У класі Program створено два статичних делегати типу Comparison<IName>:

* NameComparison: Порівнює два об'єкти IName за їх властивістю Name без урахування регістру.
* PriceComparison: Порівнює два об'єкти IName за їх властивістю Price.

Ці делегати передаються у метод Sort контейнерів при виборі відповідного критерію сортування користувачем.

Модифікація консольної програми (Program.cs):

Оновлено пункт меню для сортування (пункт 8). Тепер він пропонує користувачеві вибрати критерій сортування (за ім'ям або за ціною). Відповідний делегат (NameComparison або PriceComparison) передається у метод Sort активного контейнера.

Додано нові пункти меню (14 та 15) для тестування методів Find та FindAll. Реалізовано методи-обробники HandleFindFirstElement та HandleFindAllElements. Вони запитують у користувача критерій пошуку (за точною назвою, ціною, типом класу, вмістом підрядка в імені) та відповідне значення. На основі вибору користувача формується Predicate<IName>, який передається у методи Find або FindAll активного контейнера. Результати пошуку виводяться на консоль.

## Тестування коду

Тестування нових методів проводилося для обох типів контейнерів:Тестування Sort(Comparison<T> comparison):

Контейнери наповнювалися різними об'єктами. Викликався метод сортування з передачею NameComparison. Перевірялося, що елементи відсортовані за алфавітом імен. Викликався метод сортування з передачею PriceComparison. Перевірялося, що елементи відсортовані за зростанням ціни. Тестувалися випадки з порожнім контейнером та контейнером з одним елементом.

Тестування Find(Predicate<T> match): Здійснювався пошук першого елемента за різними критеріями: За точною назвою товару. За конкретною ціною. За назвою типу класу (наприклад, "Apartment"). Перевірялося, що повертається правильний об'єкт, або null, якщо об'єкт не знайдено.

Тестування FindAll(Predicate<T> match): Здійснювався пошук усіх елементів за різними критеріями: За точною назвою (для перевірки знаходження всіх дублікатів за ім'ям). За ціною (для знаходження всіх товарів з однаковою ціною). За типом класу (для отримання всіх об'єктів певного типу, наприклад, всіх Hotel). За наявністю підрядка в імені товару.

Перевірялося, що повертається коректна колекція знайдених об'єктів, або порожня колекція, якщо нічого не знайдено. Результати виводилися у табличному вигляді. Загальна стабільність: Перевірялася стабільність роботи програми при використанні нових функцій та їх взаємодія з існуючими операціями (додавання, видалення, серіалізація тощо).

Тестування показано на рисунках 12.1, 12.2.

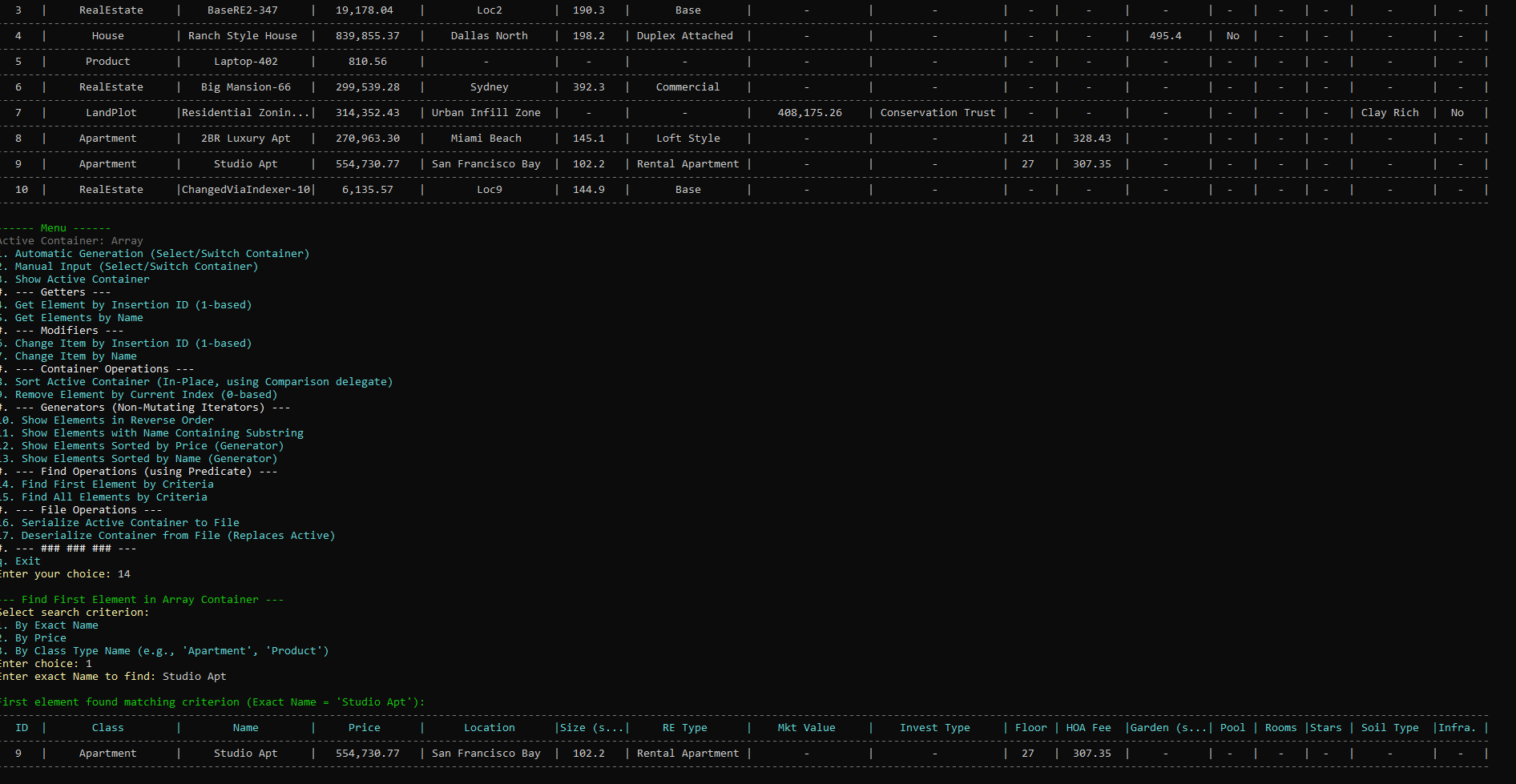


Рис. 12.1. Метод Find за iм’ям

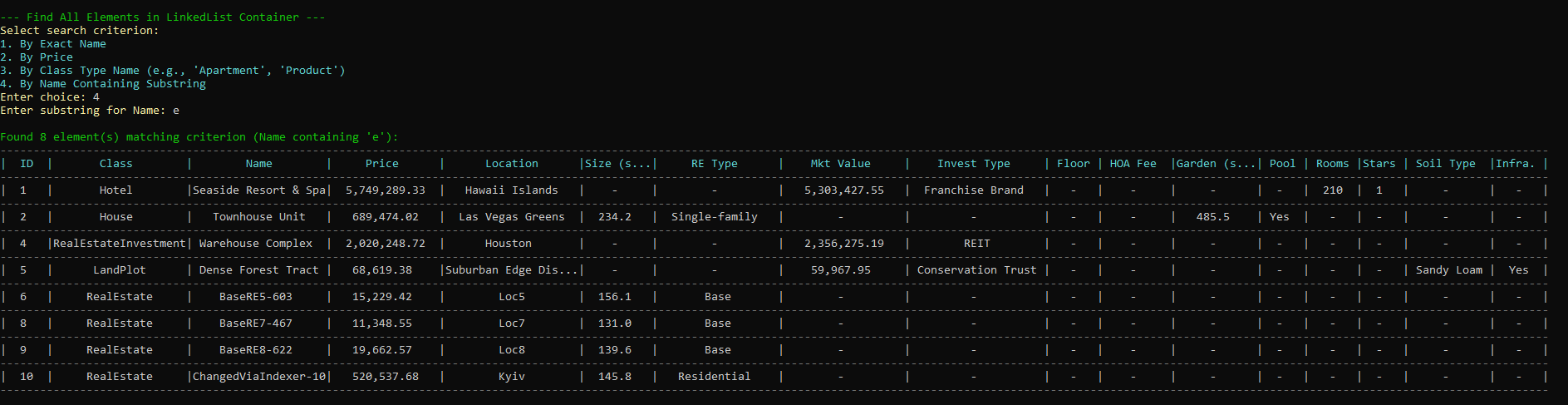


Рис. 12.2. Метод FindAll за пiдстрокою в iменi

# Використання подій для динамічного відстеження сумарної вартості у контейнерах

## Алгоритм коду

Повний код програми відображено у додатку А.

Модифікація інтерфейсу IPrice:

* До інтерфейсу IPrice додано оголошення події event EventHandler<decimal>? PriceChanged;. Параметр decimal у EventHandler призначений для передачі старої ціни товару, що дозволить контейнеру коректно розрахувати зміну загальної суми.

Реалізація події PriceChanged у класі Product:

* У класі Product властивість Price перетворено на повну властивість з приватним полем price.
* У set-аксесорі властивості Price додано логіку:
* Перевірка, чи нова ціна не є меншою або рівною нулю (генерується ValueLessThanZero, якщо так).
* Перевірка, чи дійсно змінилася ціна (this.price != value).
* Якщо ціна змінилася, зберігається стара ціна, оновлюється значення поля price, а потім генерується подія PriceChanged?.Invoke(this, oldPrice), передаючи об'єкт-відправник (this) та стару ціну.
* Реалізація відстеження сумарної вартості у Container<T> (на основі масиву):
* Додано приватне поле private decimal totalPrice; для зберігання сумарної вартості.
* Додано публічну властивість public decimal TotalPrice => totalPrice; лише для читання.
* Метод Add(T \_newObject):
  + Якщо доданий об'єкт \_newObject реалізує IPrice, його ціна додається до totalPrice.
  + Здійснюється підписка на подію \_newObject.PriceChanged методом-обробником HandlePriceChange.
* Метод RemoveById(int \_index):
  + Якщо видалений об'єкт deletedObject реалізує IPrice, його ціна віднімається від totalPrice.
  + Здійснюється відписка від події deletedObject.PriceChanged.
* Індексатор set ( this[int id] ):
  + При заміні існуючого об'єкта новим, якщо старий об'єкт реалізував IPrice, його ціна віднімається від totalPrice і відбувається відписка від його події PriceChanged.
  + Якщо новий об'єкт value реалізує IPrice, його ціна додається до totalPrice і відбувається підписка на його подію PriceChanged.
* Приватний метод HandlePriceChange(object? sender, decimal oldPrice):
  + Цей метод викликається при спрацьовуванні події PriceChanged у будь-якого об'єкта в контейнері.
  + Якщо sender є IPrice, то різниця між новою ціною (price.Price) та старою ціною (oldPrice) додається до totalPrice, таким чином коригуючи загальну суму.
* Метод PostDeserializeInitialize(): (важливо при десеріалізації)
  + Після десеріалізації контейнера цей метод викликається для повторної підписки на події PriceChanged для всіх елементів, що реалізують IPrice. Це необхідно, оскільки підписки на події не серіалізуються. Поле totalPrice при цьому не перераховується, оскільки передбачається, що ціни об'єктів вже коректні після десеріалізації, а totalPrice буде серіалізовано/десеріалізовано як частина стану контейнера (якщо це реалізовано в ContainerSerializer). Однак, у наданому коді ContainerSerializer не серіалізує totalPrice самого контейнера, тому PostDeserializeInitialize мав би також перерахувати totalPrice з нуля, або totalPrice мало б бути властивістю, що обчислюється при кожному запиті, якщо події не використовуються для цього після десеріалізації.
* Реалізація відстеження сумарної вартості у ContainerLinkedList<T>:
* Аналогічно до Container<T>, додано поле totalPrice та властивість TotalPrice.
* Методи AddFirst(T data), AddLast(T data), RemoveByIndex(int index) та set-аксесор індексатора this[int insertionIdValue] модифіковано для оновлення totalPrice та підписки/відписки від події PriceChanged об'єктів, що реалізують IPrice.
* Реалізовано приватний метод HandlePriceChange для обробки змін цін.
* Додано метод PostDeserializeInitialize() для відновлення підписок на події та, потенційно, перерахунку totalPrice після десеріалізації.
* Модифікація консольної програми (Program.cs):
* Додано новий пункт меню (18) "Show Total Price of Items in Active Container".
* Реалізовано метод-обробник HandleShowTotalPrice, який перевіряє наявність активного контейнера та виводить значення його властивості TotalPrice, відформатоване як валюта.
* При тестуванні зміни ціни об'єкта (через пункти меню 6 або 7) перевірялося, що totalPrice контейнера оновлюється коректно.

## Тестування коду

Тестування нової функціональності проводилося для обох типів контейнерів:

1. Початкова сумарна вартість: Після створення порожнього контейнера перевірялося, що TotalPrice дорівнює 0.
2. Оновлення при додаванні: До контейнера додавалися різні об'єкти (включаючи ті, що реалізують IPrice, та ті, що ні, якщо такі були б можливі). Перевірялося, що TotalPrice коректно збільшується на ціну кожного доданого товару з ціною.
3. Оновлення при видаленні: З контейнера видалялися об'єкти. Перевірялося, що TotalPrice коректно зменшується на ціну видаленого товару (якщо він мав ціну).
4. Оновлення при зміні ціни:
   * Обиравався існуючий товар у контейнері.
   * Його властивість Price змінювалася через відповідний пункт меню (який використовує рефлексію для встановлення нового значення властивості).
   * Перевірялося, що подія PriceChanged у товарі спрацювала, обробник у контейнері був викликаний, і TotalPrice контейнера коректно оновилася.
5. Отримання сумарної вартості: Багаторазово викликався пункт меню для відображення TotalPrice після різних операцій для перевірки актуальності значення.
6. Робота з різними типами контейнерів: Перевірялося, що логіка відстеження сумарної вартості працює однаково коректно як для Container<T>, так і для ContainerLinkedList<T>.
7. Десеріалізація: Після десеріалізації контейнера перевірялося, що TotalPrice відображає коректну суму (це залежить від того, чи серіалізується totalPrice самого контейнера, чи перераховується в PostDeserializeInitialize). І що подальші зміни цін об'єктів у відновленому контейнері також коректно оновлюють totalPrice.

Тестування зображення на рисунках 13.1, 13.2, 13.3.

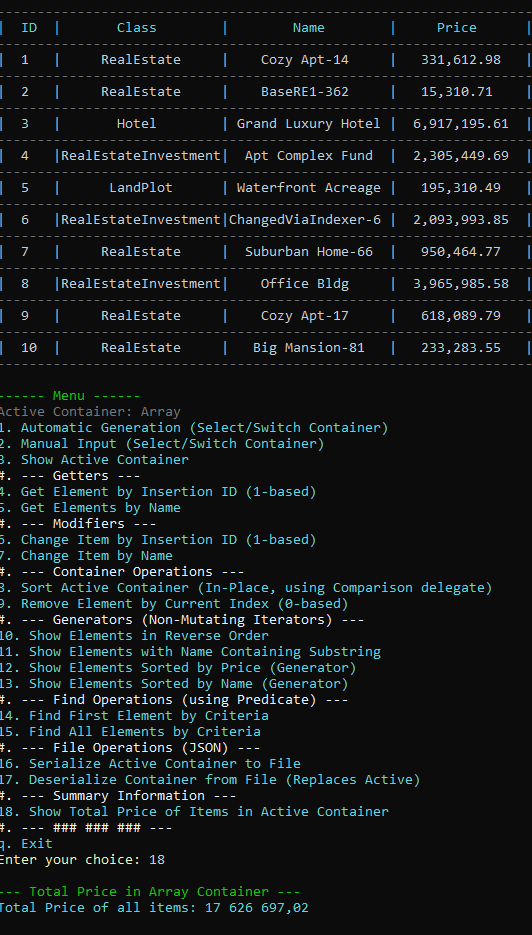


Рис. 13.1. Отримання загальної цiни

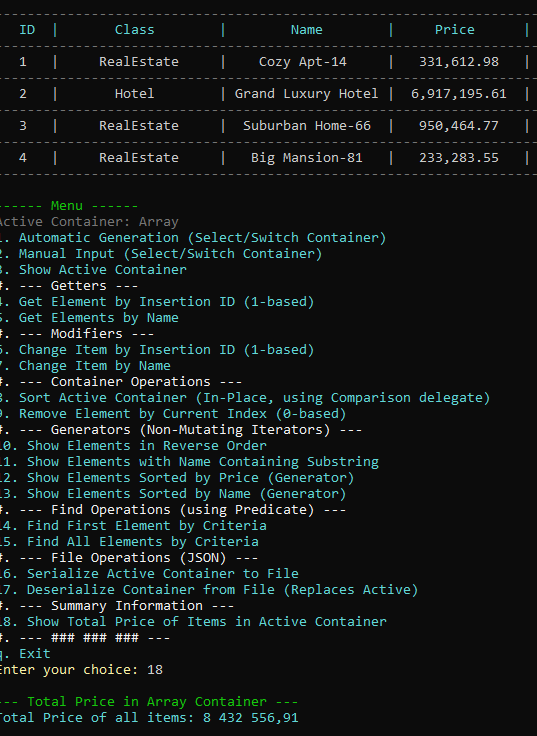


Рис. 13.2. Оновлення загальної цiни пiсля видалення товарiв

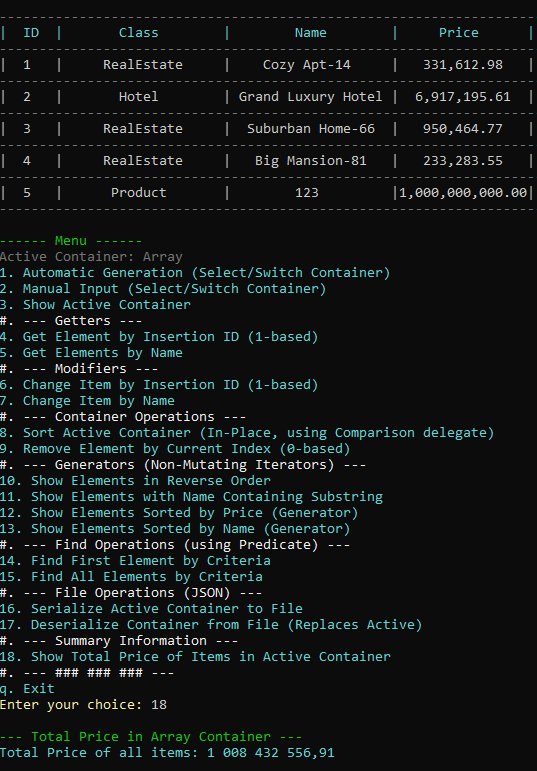


Рис. 13.3. Оновлення загальної цiни пiсля додавання товару

# Використання LINQ для аналізу даних у контейнерах

## Алгоритм коду

Повний код програми відображено у додатку А.

Для виконання завдання було додано наступні функціональні можливості до класу Program:

1. Реалізація знаходження товарів з мінімальною та максимальною ціною (LINQ):

* Створено метод-обробник HandleFindMinMaxProduct(). Цей метод спочатку перевіряє, чи активний контейнер існує та чи не порожній він.
* Далі, залежно від типу активного контейнера (containerArray або containerList), створюється IEnumerable<IName>, що містить усі елементи контейнера.
* За допомогою LINQ-методів Where, Min та Max виконується наступне:
* Фільтруються лише ті елементи, які не є null та реалізують інтерфейс IPrice (хоча в поточній реалізації всі елементи IName також є IPrice).
* З відфільтрованої колекції цін (product.Price) знаходиться мінімальне (Min()) та максимальне (Max()) значення.
* Якщо контейнер не порожній і містить товари з цінами, знайдені мінімальна та максимальна ціни виводяться на консоль.

2. Реалізація знаходження середньої вартості за категоріями (LINQ):

* Створено метод-обробник HandleFindAvarageCategoriesPrice(). Аналогічно попередньому пункту, отримується IEnumerable<IName> з активного контейнера.
* За допомогою LINQ-запиту (використовуючи синтаксис запитів або ланцюжок методів) виконується:
  + Фільтрація елементів, що не є null (та реалізують IPrice).
  + Групування елементів за типом їх класу (product.GetType().Name).
  + Для кожної групи обчислюється середня ціна (g.Average(p => p.Price)).
  + Результат проекціюється в анонімний тип або спеціальний клас/структуру, що містить назву категорії (тип класу) та середню ціну.
* Якщо контейнер не порожній, результати виводяться у табличному форматі: "Категорія | Середня Ціна".

## Тестування коду

Тестування знаходження мін/макс ціни, зображено на рисунку 1:

* Контейнери наповнювалися товарами з різними цінами, включаючи екстремальні значення.
* Викликався пункт меню 19. Перевірялося, що виведені мінімальна та максимальна ціни відповідають фактичним даним у контейнері.
* Тестувалися випадки з порожнім контейнером або контейнером без товарів з цінами (очікувалося відповідне повідомлення).

Тестування знаходження середньої ціни за категоріями, показно на рисунку 2:

* Контейнери наповнювалися об'єктами різних класів (категорій) з різними цінами.
* Викликався пункт меню 20. Перевірялося, що для кожної категорії товарів, присутньої в контейнері, виводиться коректно розрахована середня ціна.
* Тестувалися випадки, коли деякі категорії представлені одним товаром, а деякі – кількома.
* Перевірялася обробка порожнього контейнера.

Тестування зображенно на рисунках 14.1, 14.2.

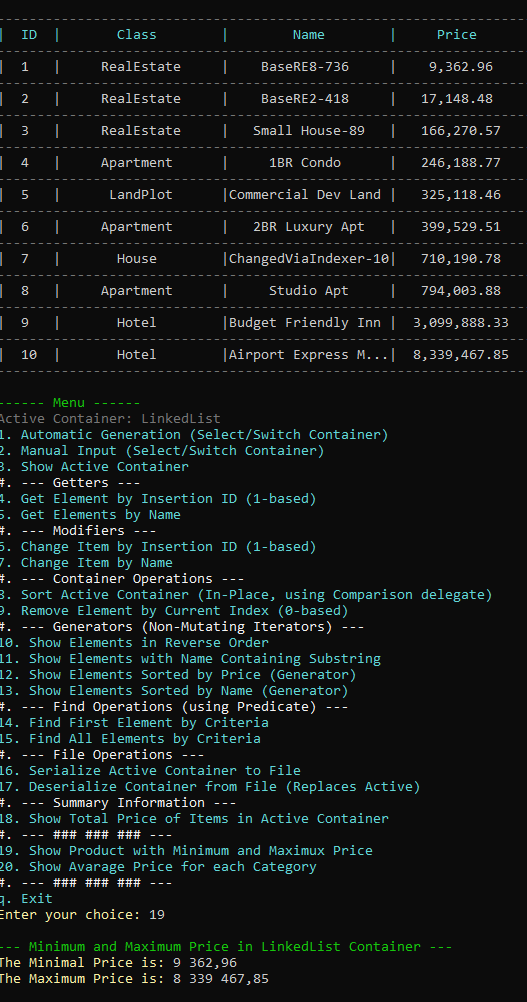


Рис. 14.1. Знаходження мін/макс ціни

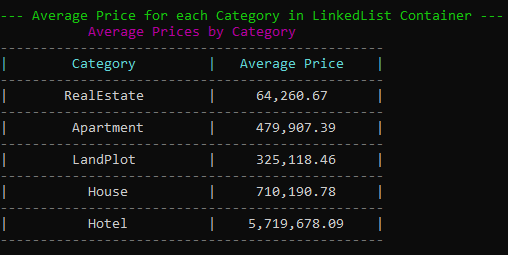


Рис. 14.2. Знаходження середньої ціни за категоріями

# Висновок

У процесі виконання комплексу лабораторних робіт було розроблено повноцінну програмну систему для управління каталогом товарів, що охопила широкий спектр фундаментальних концепцій та практичних аспектів об'єктно-орієнтованого програмування на платформі .NET з використанням мови C#.

На перших етапах було закладено основу системи шляхом проектування та реалізації гнучкої ієрархії класів для представлення різних типів товарів. Впровадження інтерфейсу IName, успадкованого від IComparable, та інтерфейсу IPrice з подією PriceChanged дозволило визначити чіткі контракти для об'єктів, забезпечивши можливість їх порівняння, сортування за іменами та динамічне відстеження змін цін. Перевантаження методу ToString() та реалізація різноманітних конструкторів підвищили зручність роботи з об'єктами.

Ключовим елементом стала розробка власних узагальнених класів-контейнерів (Container<T> на основі масиву та, опціонально, ContainerLinkedList<T> на основі двозв'язного списку) без використання стандартних колекцій .NET. Це завдання вимагало глибокого розуміння роботи з базовими структурами даних, реалізації алгоритмів додавання (з динамічним розширенням), видалення, а також індексаторів для доступу за різними критеріями (ID вставки, ім'я). Особливу увагу було приділено реалізації власних ітераторів (як з явним створенням класу-ітератора для масиву, так і з використанням самого класу списку як ітератора, обидва без yield), що забезпечило можливість використання циклу foreach та краще розуміння механізмів ітерації. Додаткові ітератори-генератори (наприклад, для пошуку за підрядком в імені) продемонстрували гнучкість yield.

Важливим аспектом стала реалізація механізмів обробки виняткових ситуацій. Створення власного класу винятку ValueLessThanZero та інтеграція блоків try-catch для обробки як стандартних, так і користувацьких винятків значно підвищили надійність та відмовостійкість розробленої системи, запобігаючи аварійному завершенню програми при некоректних діях користувача або непередбачуваних станах даних.

Реалізація власного механізму серіалізації/десеріалізації (бінарного або JSON) вмісту контейнерів дозволила забезпечити персистентність даних – можливість зберігати стан програми у файл та відновлювати його. Це вимагало розробки логіки запису/читання метаданих контейнера, типів об'єктів та їх властивостей.

Впровадження делегатних методів Sort(Comparison<T>), Find(Predicate<T>) та FindAll(Predicate<T>) суттєво розширило гнучкість контейнерів, дозволяючи передавати логіку порівняння та пошуку як параметр. Це є потужним інструментом для створення універсального коду, що легко адаптується до різних критеріїв обробки даних.

На завершальному етапі було продемонстровано використання мови інтегрованих запитів LINQ для аналізу даних у контейнерах. Реалізація запитів для знаходження товарів з мінімальною/максимальною ціною та розрахунку середньої вартості за категоріями показала переваги LINQ у декларативності, читабельності та ефективності коду для виконання складних операцій над колекціями.

Розроблена інтерактивна консольна програма слугувала ефективним інструментом для покрокового тестування та демонстрації всієї реалізованої функціональності, підтверджуючи коректність роботи кожного модуля системи.

Таким чином, виконання комплексу лабораторних робіт дозволило отримати цілісний досвід розробки програмного продукту від проектування архітектури класів до реалізації складних механізмів роботи з даними, обробки помилок та аналізу інформації. Набуті знання та навички є фундаментальними для подальшої професійної діяльності у сфері розробки програмного забезпечення.

# Додаток А

using System.Collections.Generic;

using lb\_8.Interfaces;

using System.Reflection;

namespace lb\_14