**Оцінка результатів роботи програми**

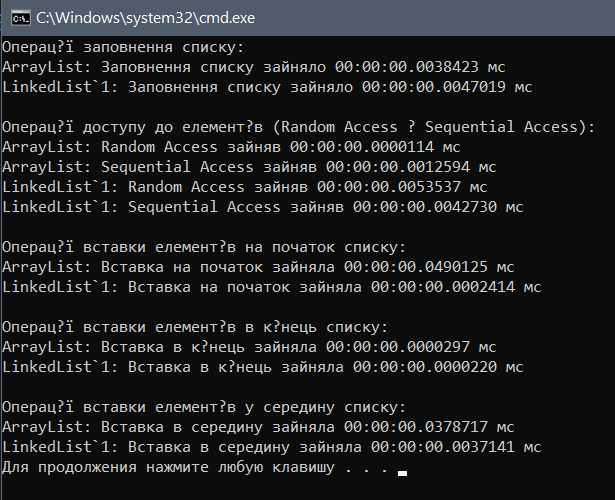
Програма написана на мові програмування C#. Тестування програми було виконано на процесорі Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz.

Була поставлена задача порівняти швидкість виконання однакових алгоритмів для **ArrayList** (Динамічний масив, який може змінювати розмір) та **LinkedList** (Зв'язаний список, елементи пов'язані між собою через посилання).

Для порівняння швидкості роботи будуть використані такі алгоритми:

1. Заповнення списку даними (методи *FillList()* )
2. Доступ до елементу Random Access (доступ за індексом) та Sequential Access (доступ за ітератором) (методи *AccessElements()* )
3. Вставка елементу на початок списку (методи *InsertAtBeginning()* )
4. Вставка елементу в кінець списку (методи *InsertAtEnd()* )
5. Вставка елементу в середину списку (методи *InsertInMiddle()* )

Після виконання програми були отриманні такі дані:



При повторних запусках програми час затрачений на виконання кожного алгоритму однаковий. Для лінійної складності час відрізняється на мікро-долі мілісекунд тому ними можна знехтувати.

Щоб спростити оцінювання результатів буду використовувати два основних позначення **O(N)** та **O(1).** В інформатиці та програмуванні O(N) та O(1) — це позначення, що використовуються в **Асимптотичній нотації** для оцінки **часової складності алгоритмів**. Вони допомагають зрозуміти, як швидко працює алгоритм залежно від розміру вхідних даних, який ми зазвичай позначаємо як N.

Якщо коротко то:

* **O(N)** означає, що час виконання алгоритму **зростає лінійно зі збільшенням кількості елементів** (вхідних даних) N. Якщо, наприклад, у нас є список із 10 елементів, а алгоритм виконує по одній операції для кожного з цих елементів, то час виконання буде залежати від кількості елементів у списку. Для 100 елементів алгоритм займе приблизно в 10 разів більше часу, ніж для 10 елементів.
* **O(1)** означає, що час виконання алгоритму **незалежний від розміру вхідних даних**. Іншими словами, алгоритм виконується за **постійну кількість часу**, незалежно від кількості елементів у масиві або списку. Це означає, що незалежно від того, скільки елементів у вхідних даних (10, 1000 або 1 000 000), алгоритм виконує фіксовану кількість операцій, яка не змінюється.

Використовуючи таку термінологію і опираючись на результати роботи програми можна зробити наступні висновки:

* **Заповнення списку**: Для обох списків має лінійну складність **O(N).**
* **Random Access**: ArrayList має **O(1)**, тоді як LinkedList має **O(N)**.
* **Sequential Access**: **O(N)** для обох структур.
* **Вставка в кінець**: **O(1)** для ArrayList (якщо не потрібно змінювати розмір) і **O(1)** для LinkedList.
* **Вставка на початок**: **O(N)** для ArrayList, оскільки потрібно змістити елементи, і **O(1)** для LinkedList.
* **Вставка в середину**: **O(N)** для обох структур.

З відси видно що ці списки майже однакові, але при великих об’ємах даних і постанові різних задач кожен з них прояве себе з різною швидкістю.

В кінці кінців можна зробити висновки про їх використання на практиці за такими критеріями:

1. **Вибір залежить від характеру операцій над даними**

* **Для частого доступу за індексом**: *ArrayList* буде більш ефективним, оскільки забезпечує швидкий O(1), доступ до елементів за індексом.
* **Для частих вставок і видалень на початку або в середині списку**: *LinkedList* підходить краще, оскільки вставка або видалення має час виконання O(1) (за умови, що є вказівник на позицію).

**Приклад**: Якщо є список замовлень, де потрібно швидко отримувати певне замовлення за індексом, то ArrayList буде ефективнішим. Однак, якщо працюємо з чергою, де постійно додаються нові замовлення на початок або видаляються з початку, LinkedList буде кращим варіантом.

1. **Об’єм даних і пам’ять**

* **ArrayList** може перевитрачати пам'ять, коли його розмір динамічно збільшується, бо при заповненні виділеної пам'яті створюється новий, більший масив, і всі елементи копіюються туди.
* **LinkedList** споживає більше пам'яті для кожного елемента через додаткові вказівники на сусідні елементи, але дозволяє уникнути перевитрат пам'яті при масштабуванні.

**Приклад**: Якщо планується робота з великою кількістю даних, які можуть часто змінюватися (наприклад, постійне додавання/видалення елементів), то LinkedList забезпечить стабільніше використання пам’яті.

1. **Продуктивність у випадках доступу і зміни**

* **ArrayList** підходить для статичних або рідко змінюваних даних, де потрібен швидкий доступ за індексом.
* **LinkedList** підходить для сценаріїв, де потрібно часто вставляти або видаляти елементи на початку чи в середині, але доступ за індексом не є критичним.

**Приклад**: У додатку для соціальної мережі LinkedList може використовуватися для зберігання коментарів у потоці, де коментарі часто додаються і видаляються, а ArrayList може підходити для зберігання профілів користувачів із рідкісними змінами.

**Загальний висновок:**

* **ArrayList** підходить, коли пріоритетом є швидкий доступ за індексом та коли змінюваність даних невелика.
* **LinkedList** краще підходить для сценаріїв із частими змінами на початку чи в середині списку, але не для прямого доступу за індексом.