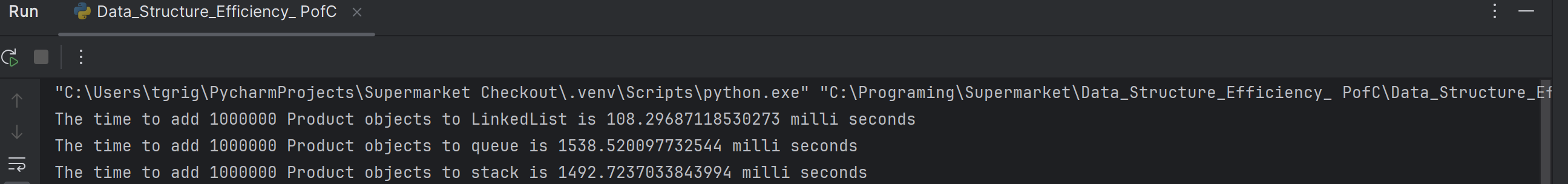
**Exploring Data Structures – Proof of Concept**

1. **Data Structure Efficiency**

Execution time in milliseconds for LinkedList, Queue, and Stack methods

****

From the above screenshot, it can be seen that the LinkedList method is the most efficient of the three methods for populating the product list. The reason for this is that LinkedLists are dynamic data structures, meaning they do not require contiguous memory allocation. When adding elements to the LinkedList, the time complexity is O(1) for insertion at the head or tail, which allows for efficient data population.

In contrast, Queues and Stacks both rely on arrays or linked lists as their underlying structure, but for the most part, they are designed for specific use cases such as managing a sequence of items in a first-in-first-out (FIFO) or last-in-first-out (LIFO) order, respectively. While both Queue and Stack structures also provide O(1) operations for adding and removing elements, they are limited by the fact that their operations only work efficiently at one end of the data structure (front or back). Therefore, while they perform well for their intended use cases, they do not offer the same level of flexibility and efficiency for bulk data insertion as a LinkedList, which can grow dynamically with minimal overhead.

1. **Sorting Algorithms Efficiency**

The efficiency of a sorting algorithm predominantly depends on its **time complexity**, which measures the number of times a particular set of instructions is executed, rather than the total runtime. This distinction is important because external factors like processor speed can influence total runtime, but time complexity provides a more accurate measure of an algorithm's efficiency.

Sorting algorithms have varying time complexities, and their performance can differ based on the initial state of the data. Here’s a description of each sorting algorithm and its performance:

* **Insertion Sort**: This simple sorting algorithm works by building a sorted section of the array one element at a time. It has an **O(n²)** time complexity in the worst case, but it performs much better on small or nearly sorted datasets, achieving a best-case complexity of **O(n)**. Its worst-case performance (O(n²)) makes it inefficient for large datasets. However, it is useful in cases where the data is nearly sorted or when only small datasets are involved.
* **Bubble Sort**: This sorting algorithm repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. It has a **worst-case time complexity of O(n²)** and performs very poorly on large datasets. While it does have a best-case time complexity of **O(n)** when the data is already sorted, its constant O(n²) behavior makes it inefficient for large volumes of data.
* **Selection Sort**: This algorithm works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and swapping it with the element at the beginning (or end). Like Bubble Sort, **Selection Sort** has a worst-case time complexity of **O(n²)**, and while it’s a bit more efficient than Bubble Sort due to fewer swaps, it’s still inefficient for large datasets.
* **Merge Sort**: Merge Sort uses a divide-and-conquer approach to split the data into smaller sub-arrays, sort each sub-array, and then merge them back together in sorted order. It has a **time complexity of O(n log n)** in both the best and worst cases, making it much more efficient than algorithms like Bubble Sort, Insertion Sort, and Selection Sort, especially for large datasets. Its time complexity remains predictable regardless of the initial data order, making it ideal for large volumes of data that need to be sorted efficiently.

Given the above descriptions, **Merge Sort** is the most efficient for sorting large datasets due to its consistent O(n log n) time complexity, even in the worst-case scenario. For a system like a supermarket, where sorting large datasets frequently may be required, Merge Sort is the most practical choice, ensuring efficient performance regardless of the initial dataset order.

**References**

Board Infinity. (n.d.). *Time complexity of sorting algorithms*. Board Infinity. <https://www.boardinfinity.com/blog/time-complexity-of-sorting-algorithms/>  
  
GeeksforGeeks. (n.d.). *Time complexities of all sorting algorithms*. GeeksforGeeks. <https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/>

1. **Searching Algorithms Efficiency**

Execution time in milliseconds for Linear search and Binary Search

A screen shot of a computer

Description automatically generated

From the above screenshot, it can be seen that **Binary Search** is significantly more efficient than **Linear Search** when finding a specific product in a large list of 1,000,000 products. Here’s why Binary Search outperforms Linear Search:

* **Linear Search**: This algorithm checks each element in the list one by one, starting from the first element and continuing until the target element is found. In the worst case, it must check every element, resulting in a **time complexity of O(n)**. For large datasets, this becomes inefficient because the algorithm must potentially examine every single item in the list.
* **Binary Search**: Binary Search, on the other hand, works on **sorted data** by repeatedly dividing the search interval in half. It starts with the entire list and compares the target value with the middle element. If the target value is smaller, the search continues on the left half; otherwise, it continues on the right half. This process continues until the target is found or the interval is empty. **Binary Search** has a time complexity of **O(log n)**, which is much faster than Linear Search, especially when dealing with large datasets. For example, with 1,000,000 products, Binary Search will perform at most 20 steps, while Linear Search may require up to 1,000,000 steps.

Therefore, **Binary Search** is far more efficient than **Linear Search** for large datasets due to its logarithmic time complexity, significantly reducing the number of comparisons needed to find the target item.