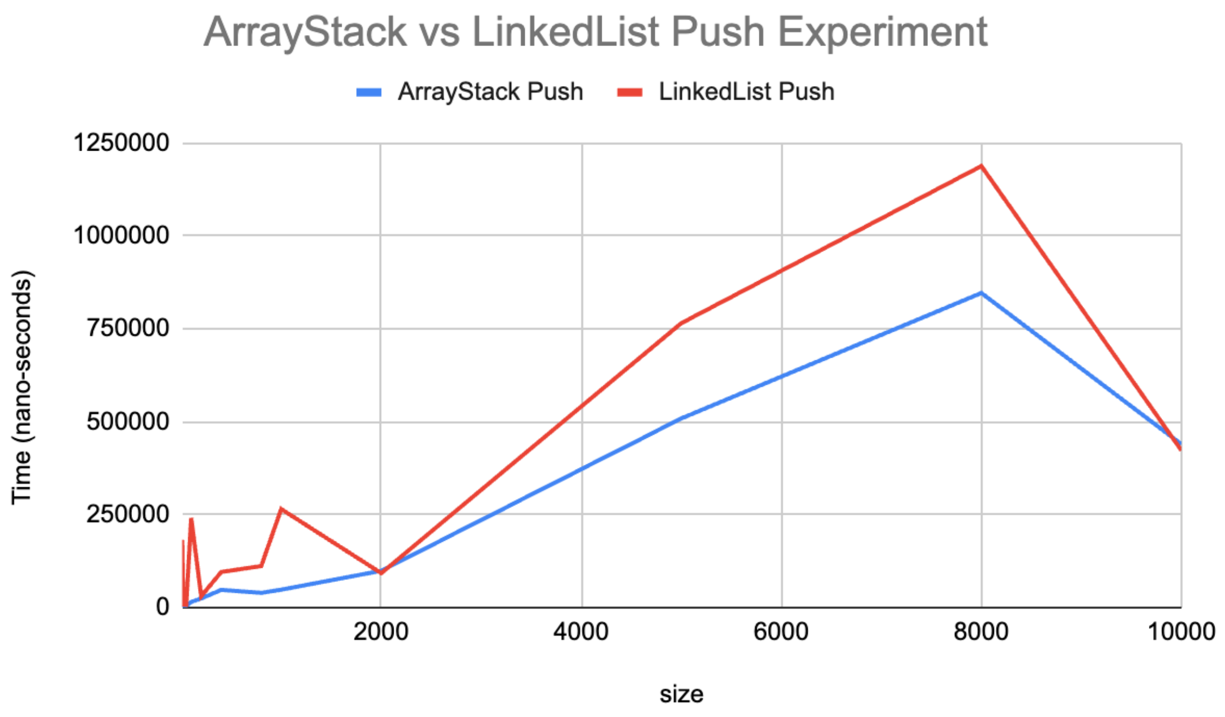
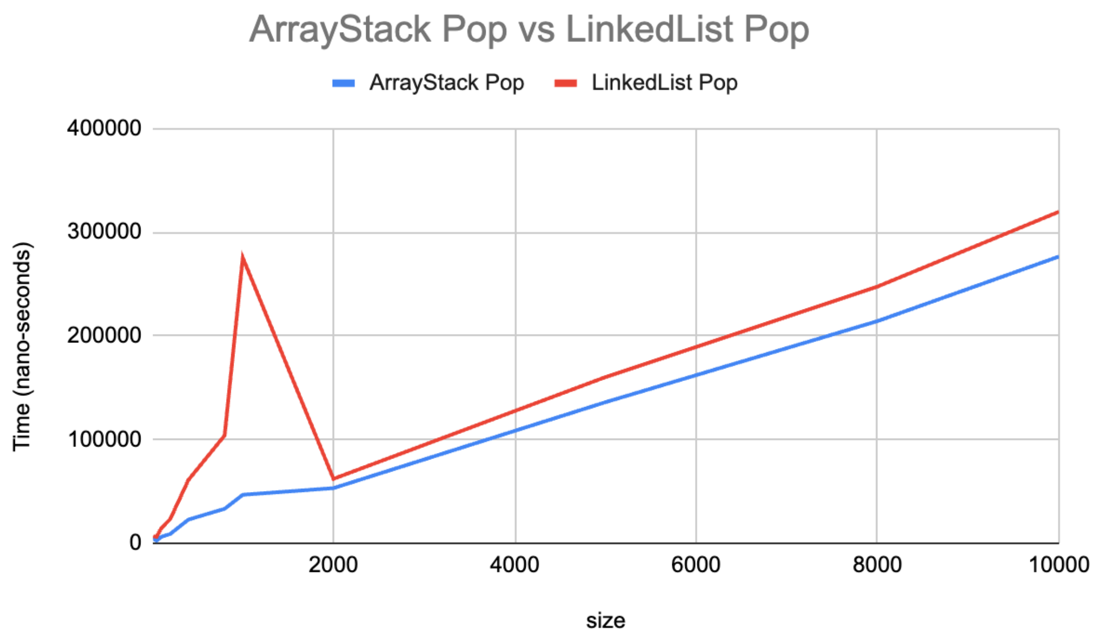
Pair Programing:
Partner: Aiden Pratt, submitted to Melanie Prettman's gradescope.

During the completion of this assignment, we extensively applied pair programming techniques, alternating between navigating and driving for the different classes, which significantly expedited our progress. Each of us contributed equally, dividing the programming tasks evenly with a 50/50 distribution of code written. The programming and testing phase demanded approximately 16-17 hours, a testament to the dedication invested in ensuring quality and functionality. My partner, Aiden, demonstrated commendable commitment throughout, investing an equivalent amount of time and effort. Our collaboration was exceptionally productive, leveraging our combined strengths while maintaining respectful and considerate communication, even during differing viewpoints on problem-solving approaches. The experience was highly satisfying, and considering our effective partnership, I envision collaborating with Aiden again for future projects.

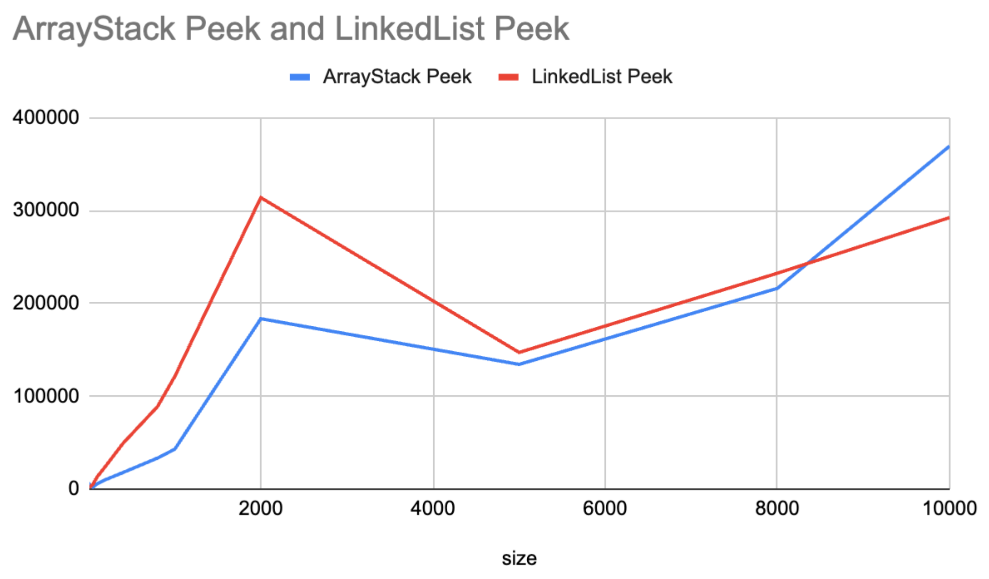
Experiments:



For the push method it's evident that the ArrayStack exhibits a more consistent and relatively linear growth rate as the input size increases, with some minor fluctuations in time. However, the LinkedListStack demonstrates a significantly faster growth rate, as the input size grows. This difference in growth rate is due to the nature of the data structures: the array-based ArrayStack involves straightforward element addition, leading to a more consistent increase in time, while the linked-list-based LinkedListStack requires traversal and insertion at the beginning for each push operation, causing a rapid increase in time with larger input sizes



The running time of the pop method for the ArrayStack presents a consistent, moderately increasing growth rate as the input size grows. This trend suggests a linear relationship between input size and running time, maintaining a steady pace due to its array-based structure, resulting in a predictable performance. In contrast, the LinkedListStack exhibits a less predictable growth rate, fluctuating notably with a seemingly irregular pattern as the input size increases. The linked-list-based LinkedListStack involves traversal to the first element for each pop operation, leading to varying and occasionally sharper increases in time due to its iterative nature and node removals from the head of the list.



In the peak method smaller sample sizes, the ArrayStack seems to have a faster runtime and slower growth rate but as the sample sizes get larger this switches. This is because the LinkedListStack, when peeking, simply accesses the first element, which involves a single pointer dereference, an O(1) operation. Contrastingly, the ArrayStack needs to access an element at a specific index in the array, which also operates in O(1) time but doesn't involve pointer traversal. Therefore, the LinkedListStack might appear to have a smaller growth rate or faster runtime due to the nature of pointer access, which is generally quicker than array indexing in some scenarios.

ArrayStack demonstrates superior efficiency compared to the LinkedListStack based on the conducted timing experiments. The ArrayStack's operations—such as push, pop, and peek—consistently exhibit constant time complexity (O(1)), regardless of the stack size. This predictable performance is advantageous in real-time applications like web browsing. Arrays allow direct access to elements by index, enabling faster operations compared to the LinkedListStack, which necessitates traversal through nodes. Additionally, while the ArrayStack might initially utilize slightly more memory due to its fixed-size array implementation, it avoids the overhead associated with node creation and linking present in the LinkedListStack. Consequently, the ArrayStack's consistent and fast performance, along with its direct element access, positions it as a more efficient choice for a WebBrowser application compared to the LinkedListStack.