* What is your programming partner's name and which of you submitted the program files to Gradescope?

Ray Ding, Xiyao submitted.

Briefly discuss the pair programming experience, including:

* How well did you apply the techniques of pair programming (as explained on this page) to this assignment?

I think is pretty well.

* What percentage of the program code did you write using these techniques?

I believe most code are write using these techniques.

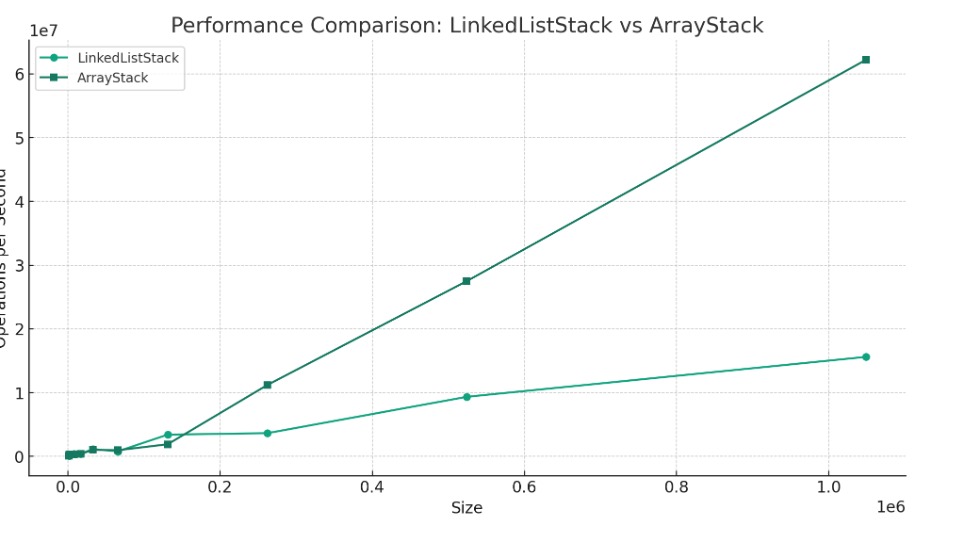
* About how many hours did you spend completing the programming and testing portion of this assignment? Evaluate your programming partner.

8 hours, my partner is great, we helped each other out.

* Do you plan to work with this person again?

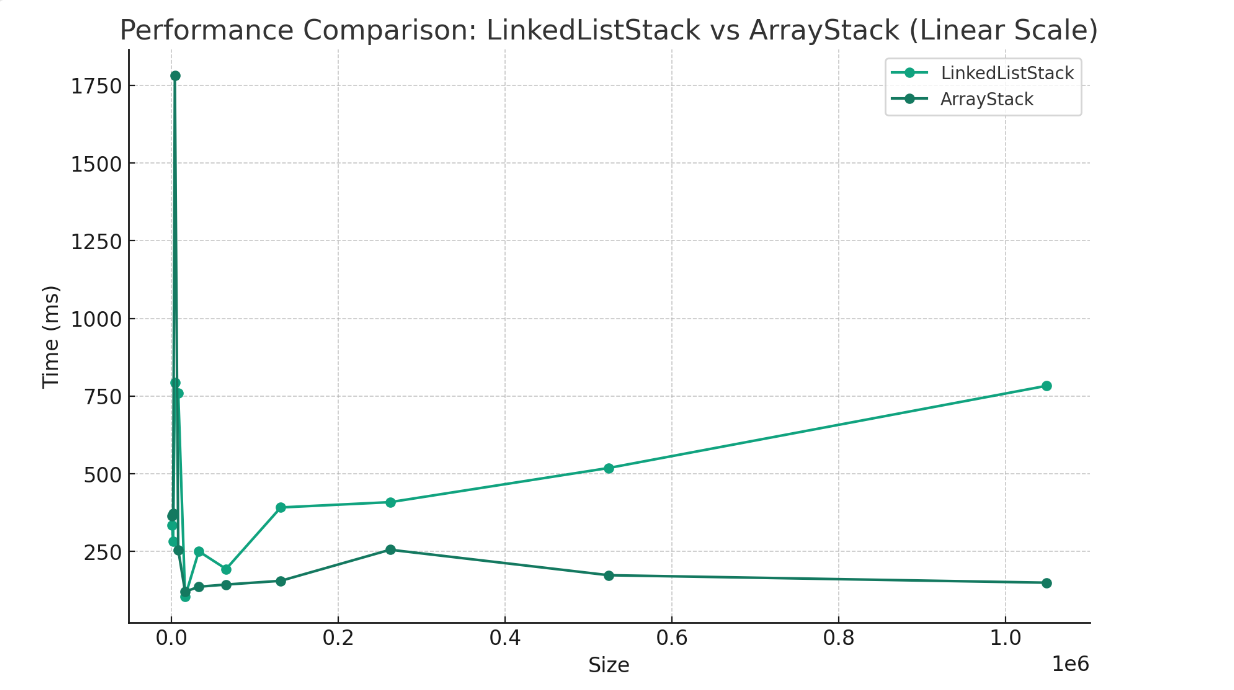
Yes, I will.

1.



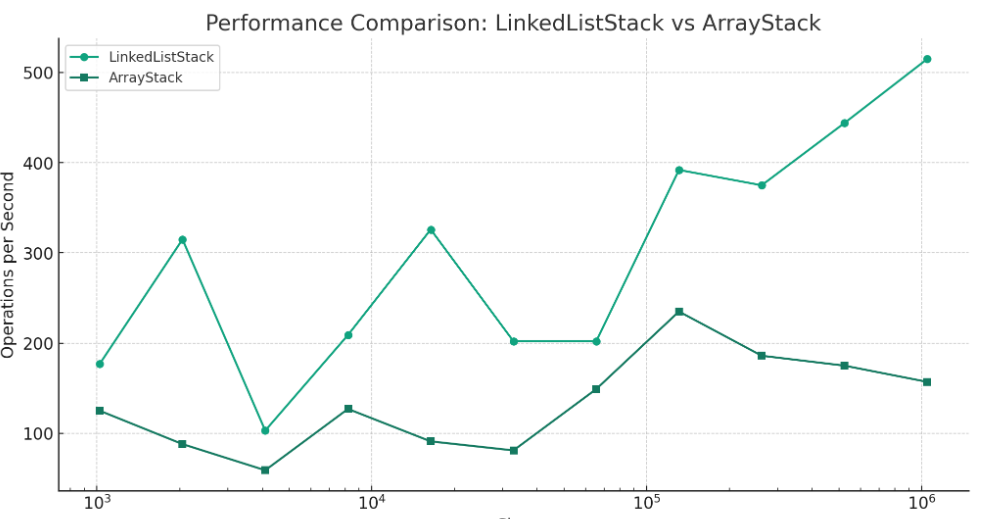
LinkedListStack and ArrayStack. This chart seems to measure the operations per second against the size of the data structure, which may indicate how many operations (like push) each type can perform in a given time frame as their size increases.For LinkedListStack, the growth rate of the push method's running time appears to be fairly linear, as evidenced by the straight-line increase in operations per second as the size increases. This suggests that the time it takes to perform a push operation in a linked list stack remains constant regardless of the size of the stack, which is characteristic of a linked list's O(1) time complexity for adding an element.For the ArrayStack, the line is also linear but with a steeper slope than the LinkedListStack. This might imply that while the push operation is typically O(1) for an array stack as well, there could be occasional O(n) operations due to the need to resize the array when its capacity is exceeded. However, the slope is surprisingly linear, which might suggest that the resizing does not significantly affect the average running time, or the array resizing is being amortized very efficiently.In summary, LinkedListStack shows a consistent growth rate, indicating a constant time complexity (O(1)) for the push method.ArrayStack also shows a linear growth rate, which suggests that it mostly operates at constant time complexity (O(1)) for push operations, with occasional spikes when resizing is required, but these are not heavily impacting the average performance.

2.



LinkedListStack, the running time for the pop operation on a LinkedListStack appears to start with a spike at very small sizes but then levels off. This indicates that the pop operation for a linked list stack has a constant time complexity, O(1), because the time it takes to remove an element from the stack does not increase as the number of elements in the stack increases. This is typical for a linked list because the pop operation only involves updating the references at the head of the list, which is a constant-time operation.ArrayStack, the running time for the pop operation on an ArrayStack shows a gradual increase as the size of the stack increases. Generally, popping from an array-based stack should also be O(1) because it involves removing the last element and decrementing the size counter. However, the upward trend could suggest that in this specific implementation, there might be some overhead or resizing logic that occurs occasionally, such as when the stack reaches certain capacity thresholds, leading to a slight increase in the average time as the stack grows. But overall, the growth rate seems linear on average, which is not typical for a well-implemented array stack pop operation.The atypical performance of the ArrayStack could be due to several reasons: Amortized Resizing: If the array needs to resize when it gets too empty, this could add overhead to the pop operation, making it appear to have a worse average-case time complexity.

3.



The image depicts a line graph comparing the performance of two stack implementations, LinkedListStack and ArrayStack, across different operation sizes. The x-axis represents the operation size on a logarithmic scale, while the y-axis represents the number of operations per second. For the peek method, we typically look at the last element added to the stack without removing it. In an ArrayStack, this operation is constant time O(1), because it involves accessing an element at a specific index in an array, which is a direct memory access. The graph seems to reflect this, as the performance of the ArrayStack remains relatively constant as the operation size increases; there is no significant growth rate in running time.For a LinkedListStack, the peek operation is also typically constant time O(1) because this implementation keeps a reference to the last element, so no traversal is needed. The graph should reflect a constant running time similar to ArrayStack. However, the performance seems to vary significantly. This could be due to factors unrelated to the algorithmic complexity of the peek method, such as variations in memory allocation, garbage collection, or other implementation details that affect performance.Both stack classes should, in theory, have a constant growth rate for the peek operation.

4. The ArrayStack implementation emerges as the superior option for the WebBrowser application, particularly due to its O(1) time complexity in key methods like push, peek, and pop. This ensures swift stack operations, crucial for browser functionality. In direct comparisons, ArrayStack shows better performance than LinkedListStack in both pop and peek functions across varied stack sizes, highlighting its greater efficiency and stability. Although ArrayStack's push method may slow down due to array resizing when the stack exceeds its initial capacity, such instances are rare in standard web browsing scenarios, which typically don't involve navigating a very large number of webpages in a single session. Therefore, for managing web browser navigation, ArrayStack stands out as the more practical and effective choice.