CSE 332: Data Structures and Parallelism

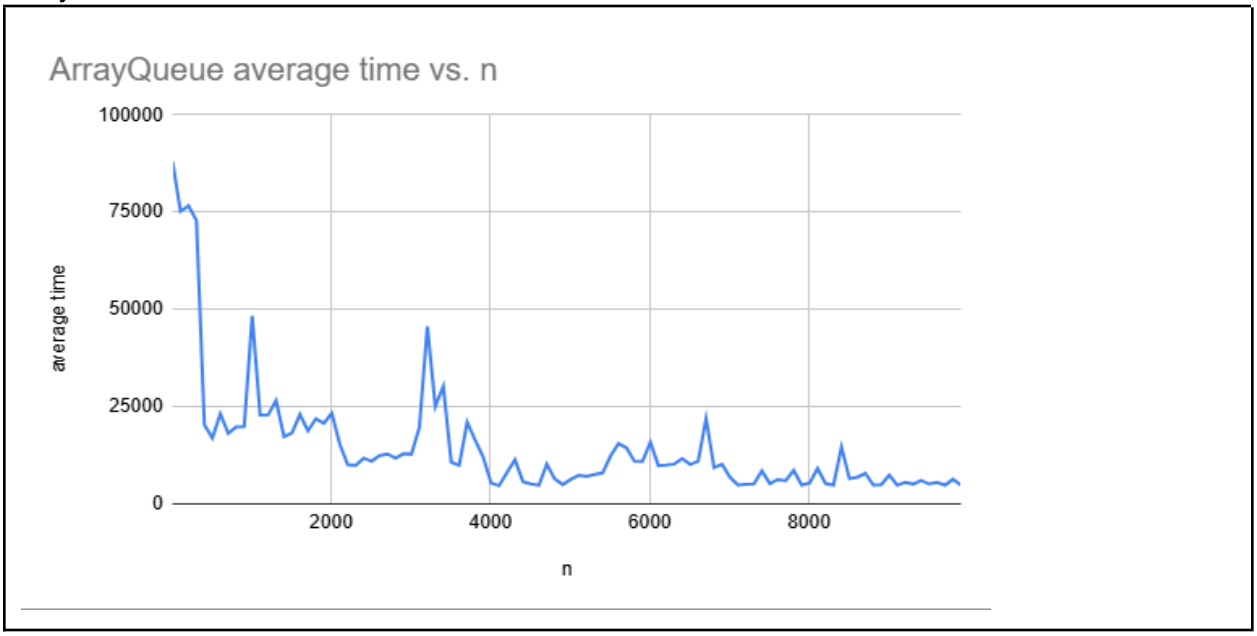
Exercise 0 - Benchmarking Worksheet

Exercise 0 Benchmarking Worksheet (25wi)

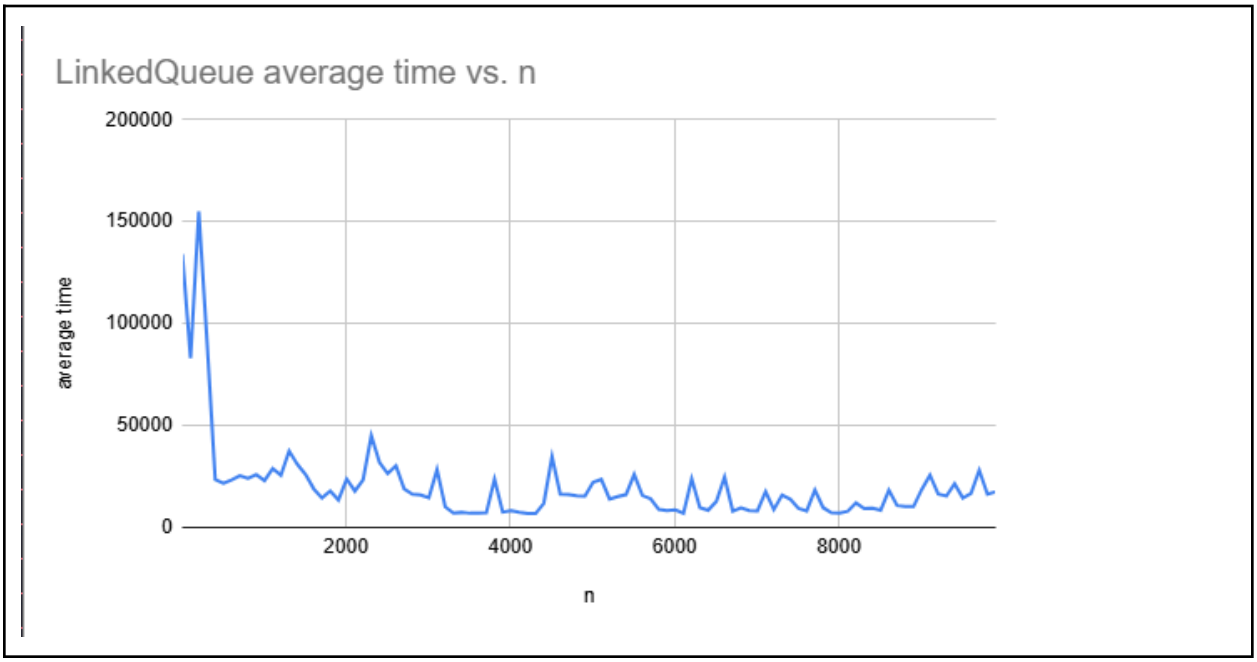
Question 1:

For all of the future questions we will need to compare the benchmarking results for the 6 classes ArrayQueue, LinkedQueue, ArrayStack, LinkedStack, NaiveQueue, and NaiveStack. Include your charts in the corresponding box below:

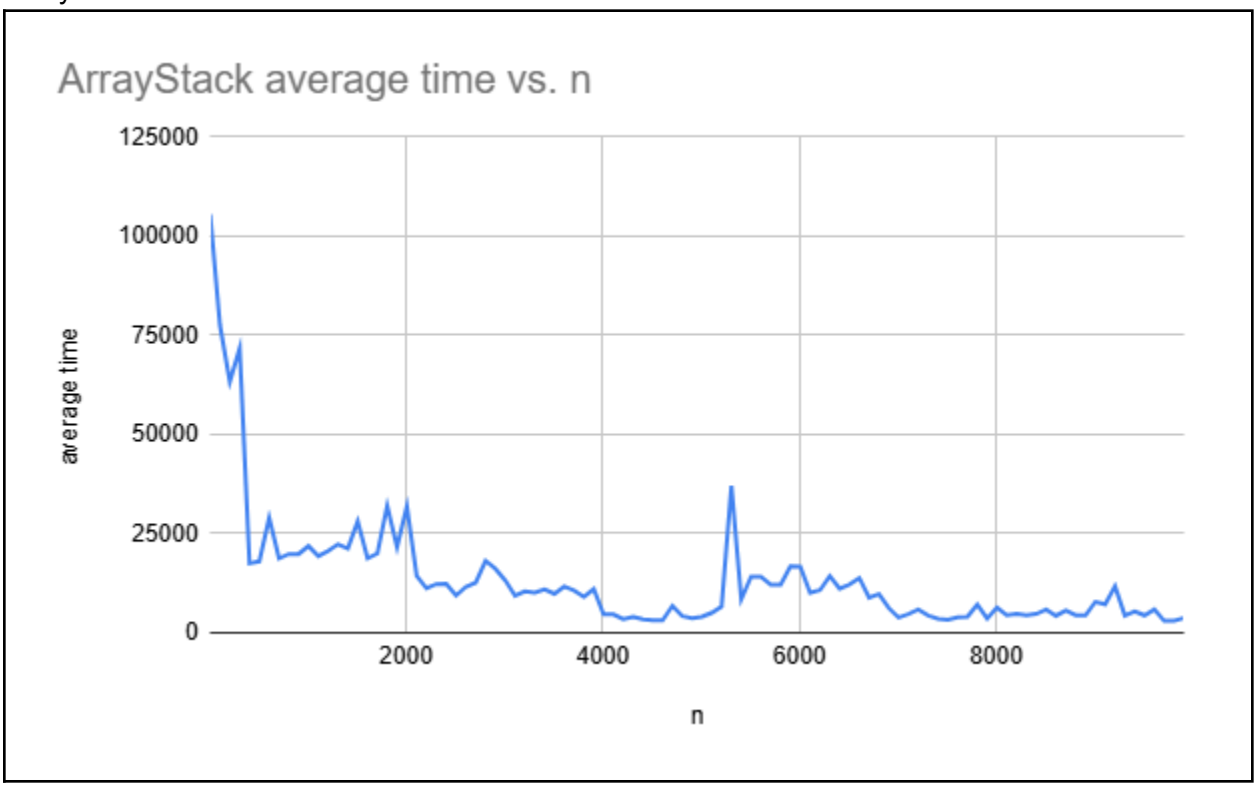
ArrayQueue



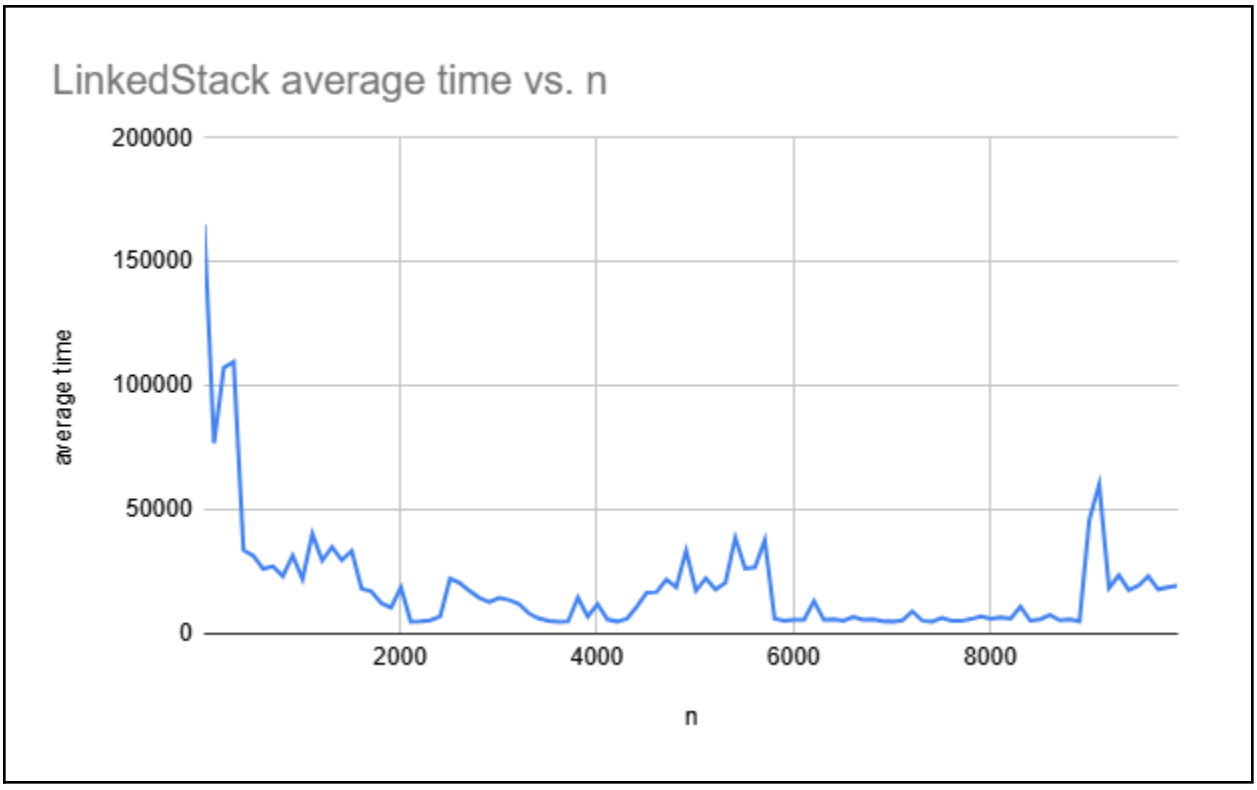
LinkedQueue



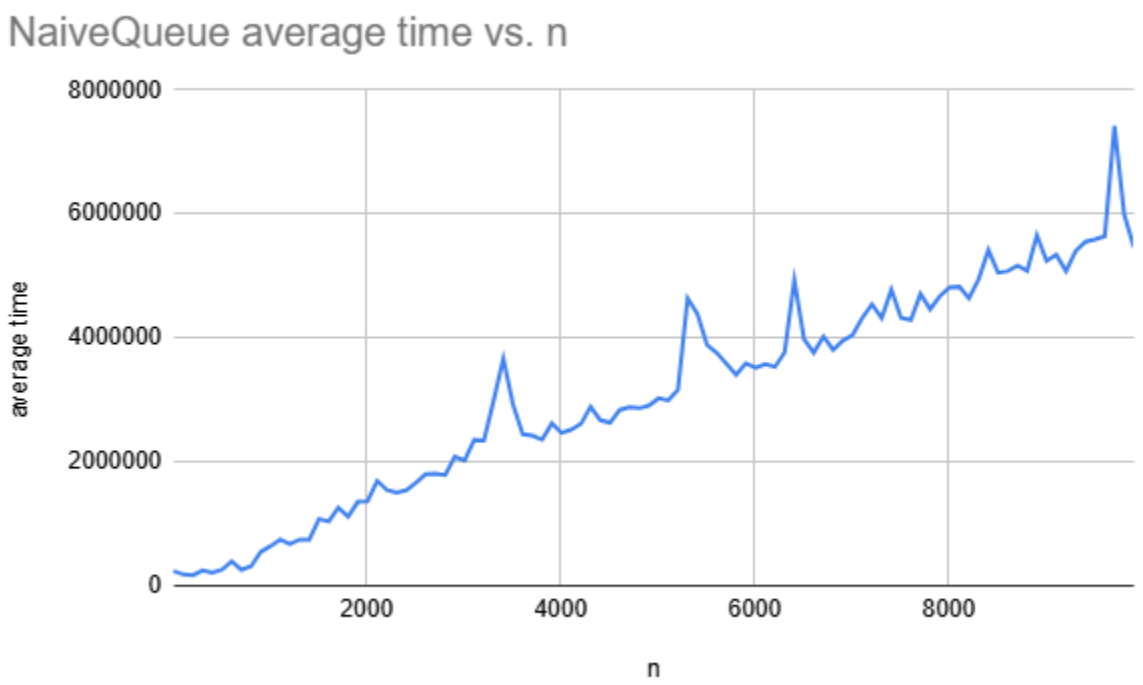
ArrayStack



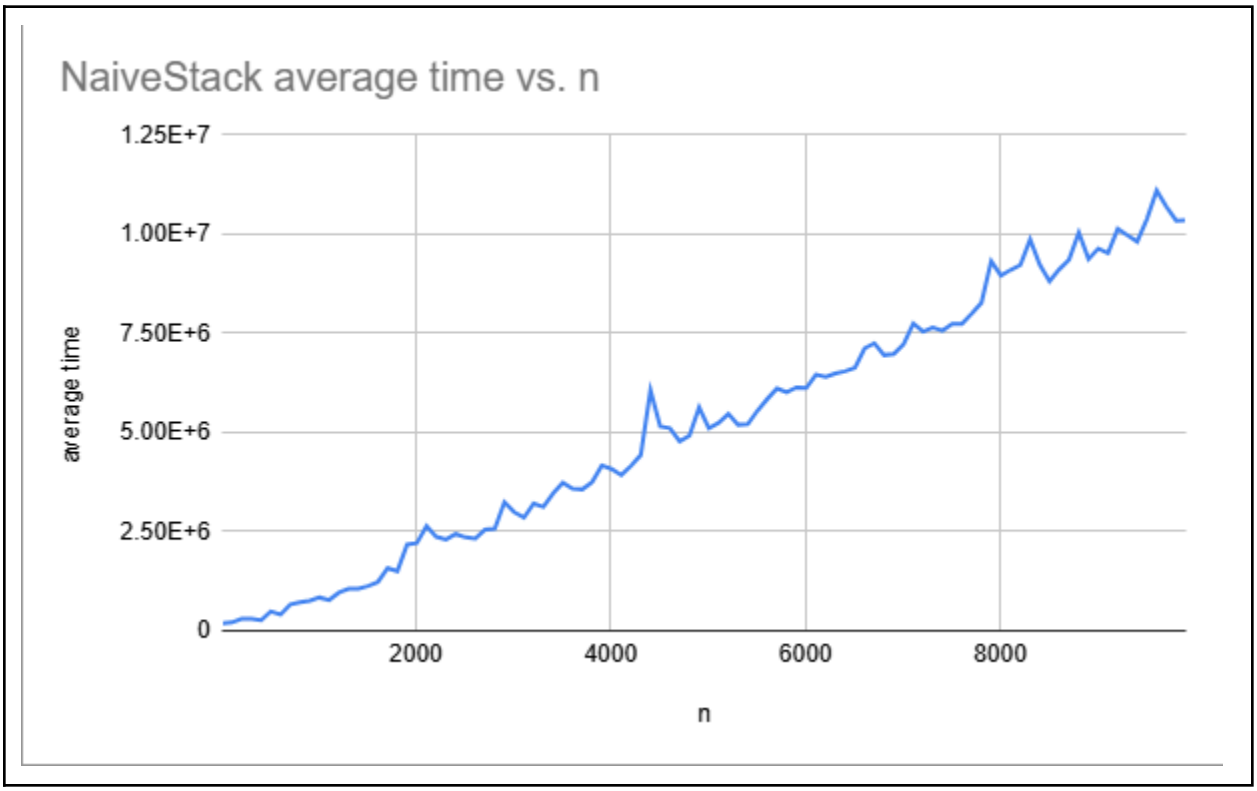
LinkedStack



NaiveQueue



NaiveStack



Question 2:

If all went according to plan, all four of the data structures that you implemented should run in constant time. This should be evident in your charts above by noting that the running time trends to a horizontal line. By comparison, the running times of NaiveQueue and NaiveStack increase as n gets larger, meaning that the running time is worse than constant. In fact, the running time of adding plus removing is linear, i.e. O(n). For each of these classes at least one of enqueue/dequeue or push/pop is linear time. For this question, reference the implementations of these classes, identify which operation is linear time (or else say both are), and justify why it runs in linear time (1-2 sentences should be sufficient).

NaiveQueue - Which operation is linear time? (enqueue, dequeue, or both)

dequeue

Justification:

The remove(0) method of ArrayList in dequeue() requires shifting all subsequent elements in the list one position to the left. This shifting involves traversing all elements after the first one, resulting in an O(n) operation for a list of size n; enqueue() time complexity is O(1), because adding an element to the end of an ArrayList is an efficient operation that typically does not involve shifting elements.

NaiveStack - Which operation is linear time? (push, pop, or both)

both

Justification:

stack.add(0, value) in push requires inserting an element at the beginning of the ArrayList, which involves shifting all existing elements one position to the right. This traversal of all elements results in O(n); stack.remove(0) in pop() requires removing the first element from the ArrayList, which involves shifting all subsequent elements one position to the left which is also O(n).

Question 3:

For this question, we will look at why it's important to double the size of the array when resizing in the array-based data structures.

First, modify your ArrayStack data structure so that instead of doubling the size of your array at resizing, you add 5 to the size of the array. After making this change, run the benchmark again for ArrayStack. At this point, you should notice that the benchmark takes SUBSTANTIALLY longer (run your code long enough to observe this, but it will be painful if you wait for it to finish). Summarize why you believe the running time was so much worse! (1-2 sentences should suffice)

If the array size is increased by only 5 instead of doubling during resizing, the resizing operation will occur much more frequently as the number of elements grows. Each resizing requires copying all existing elements to the new array, leading to a total time complexity of O(n^2) instead of the amortized O(1).