CSE 332: Data Structures and Parallelism

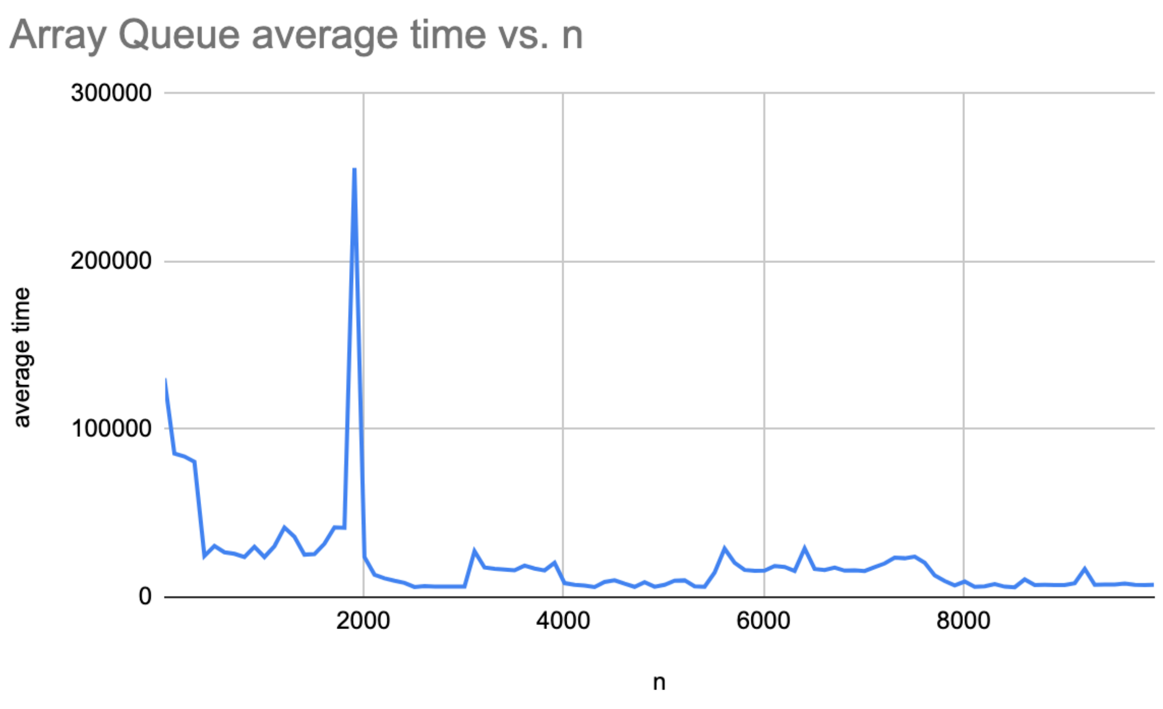
Exercise 0 - Benchmarking Worksheet

Exercise 0 Benchmarking Worksheet (25wi)

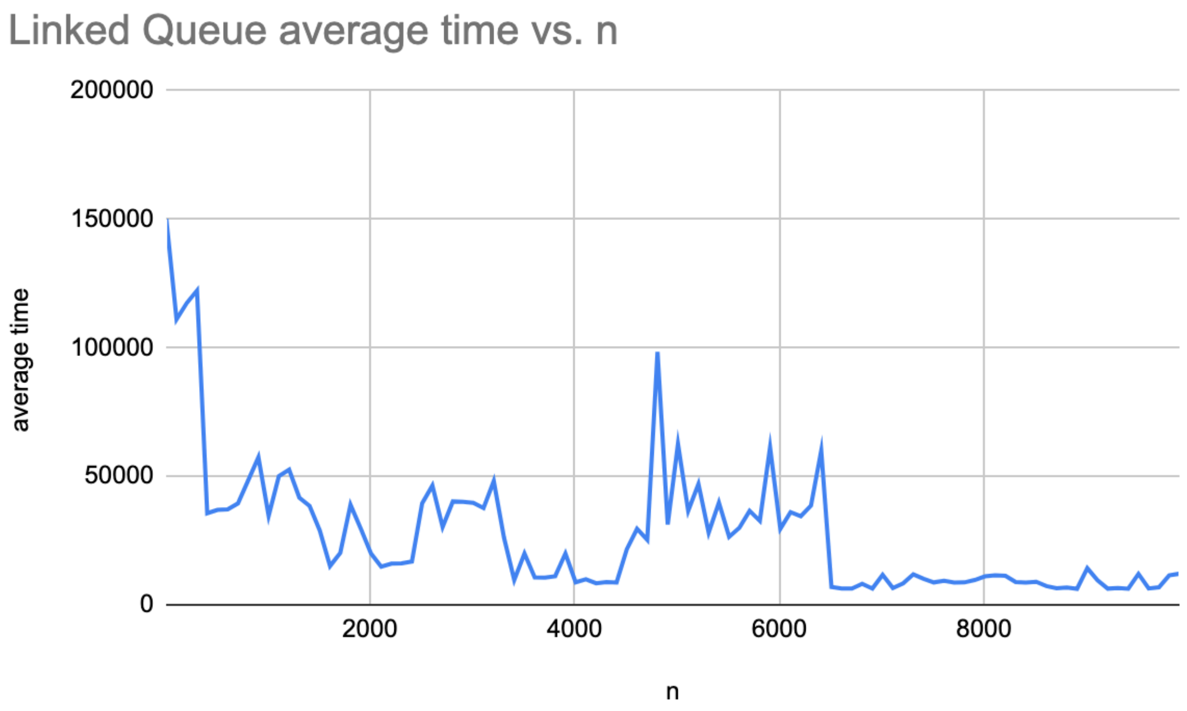
Question 1:

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:

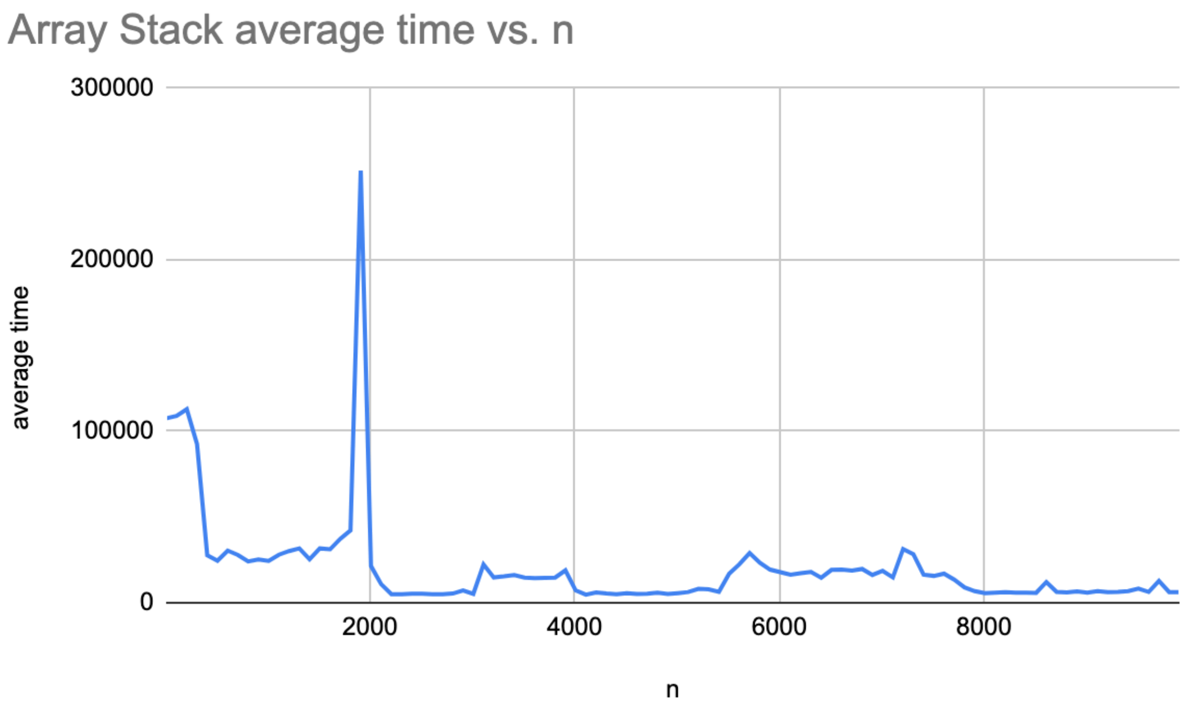
ArrayQueue



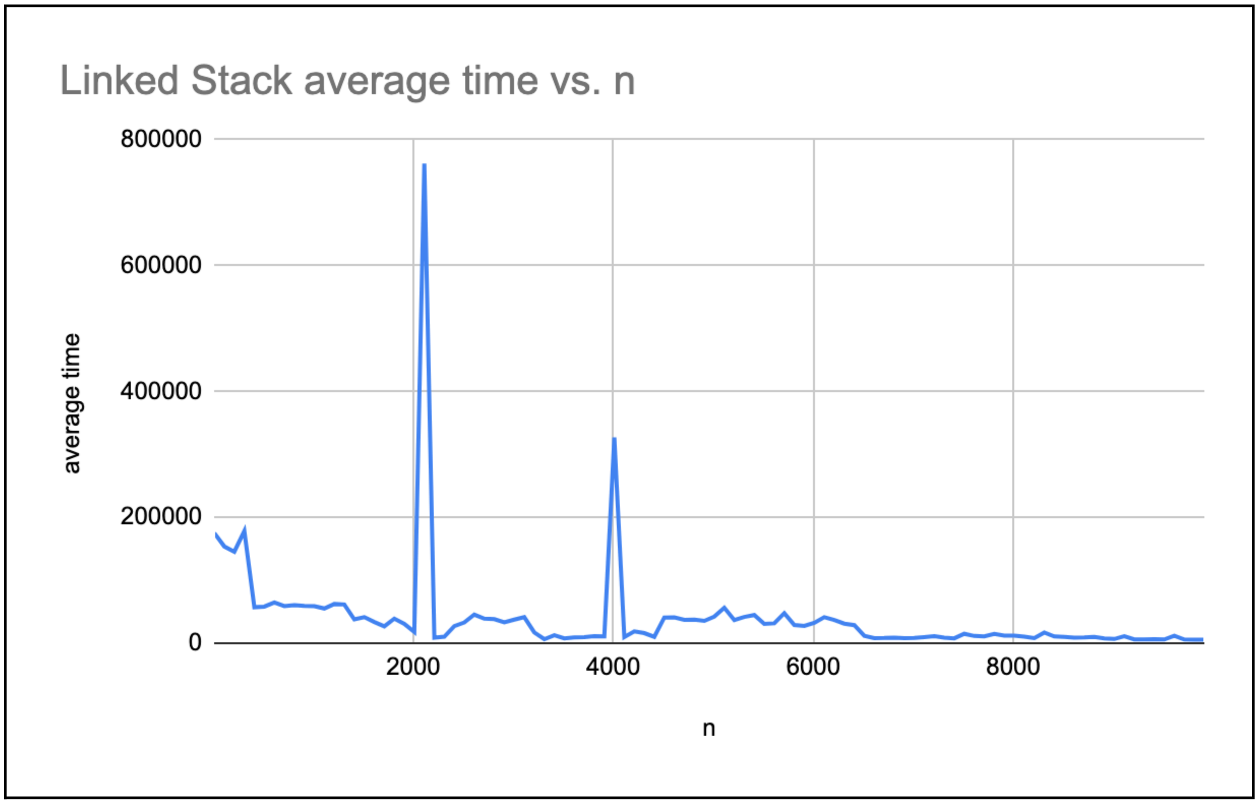
LinkedQueue



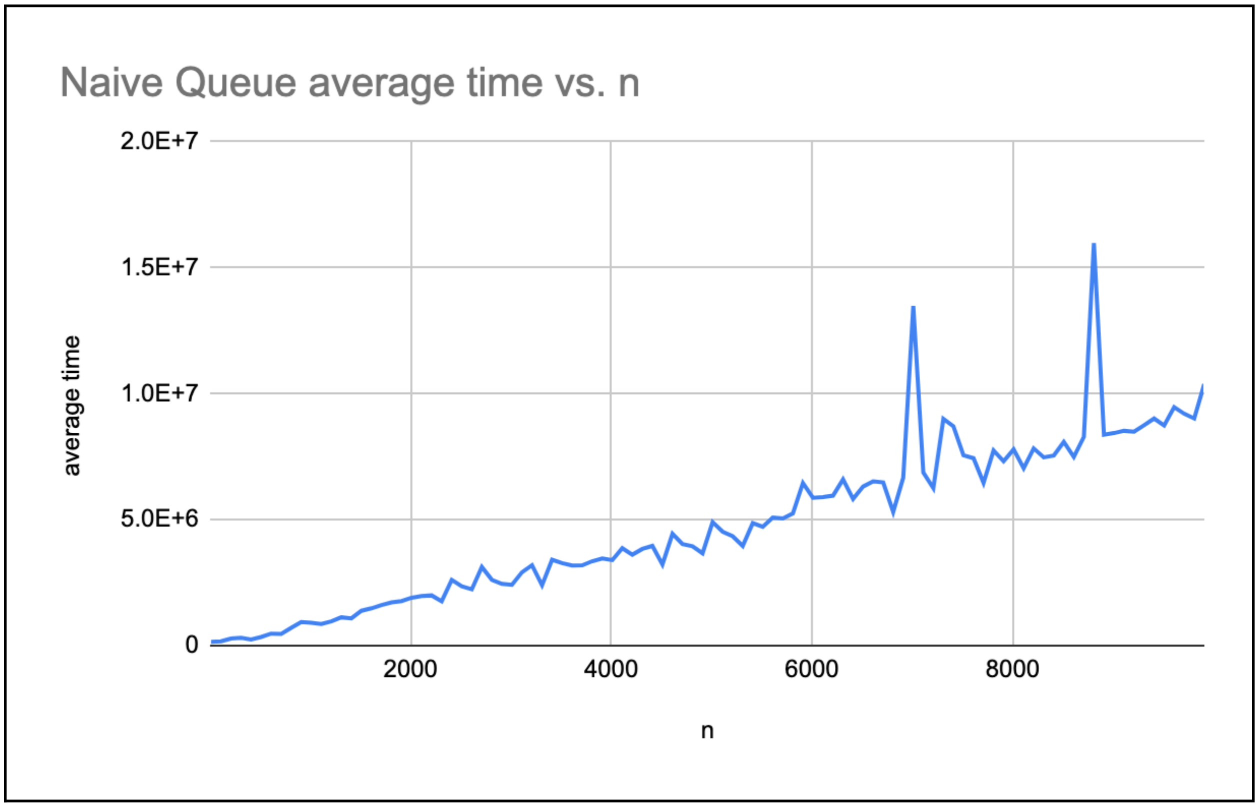
ArrayStack



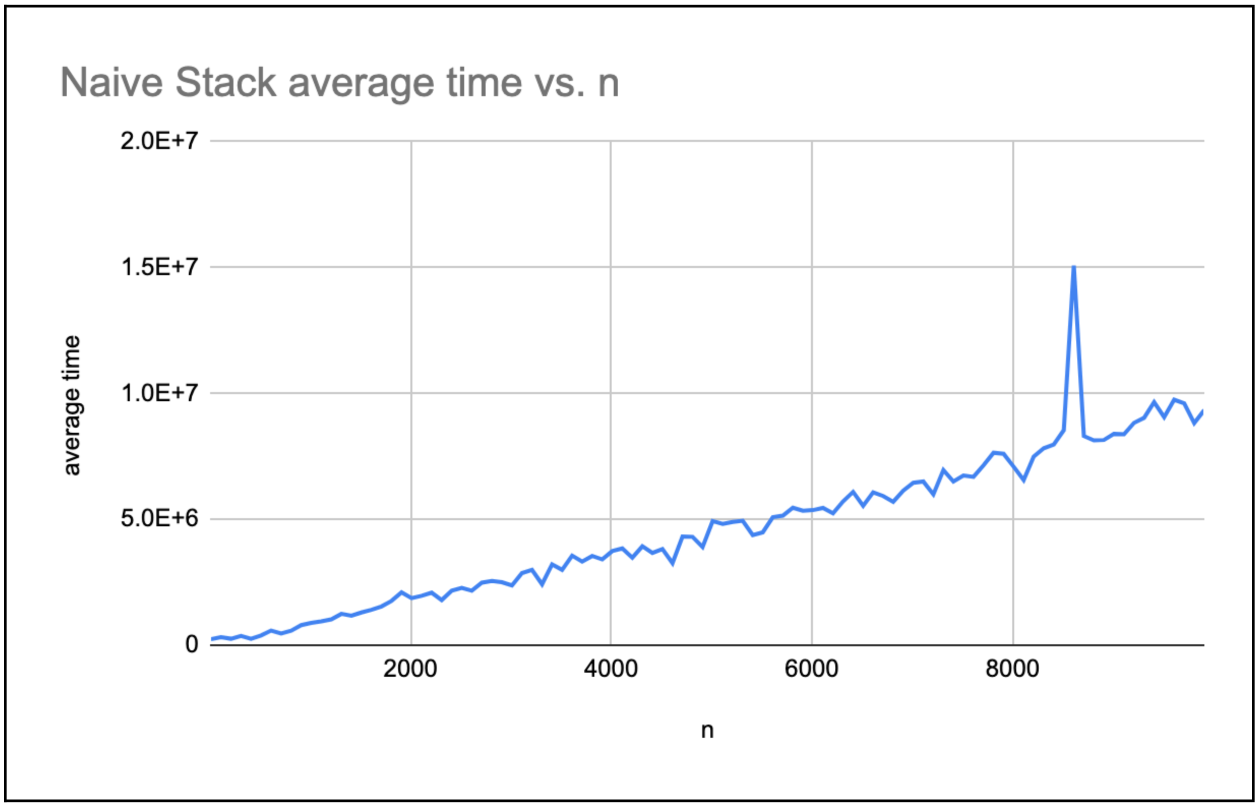
LinkedStack


Linked Stack average time vs. n
800000
600000
400000
200000
0
2000
4000
6000
8000
average time
n

NaiveQueue


Naive Queue average time vs. n
2.0E+7
1.5E+7
1.0E+7
5.0E+6
0
2000
4000
6000
8000
average time
n

NaiveStack


Naive Stack average time vs. n
2.0E+7
1.5E+7
1.0E+7
5.0E+6
0
average time
2000
4000
6000
8000
n

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:

Because the `.remove` in dequeue removes the element at 0 position and shifts all the subsequent elements one position to the left. The shifting operation takes O(n) time.

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

both

Justification:

Because the `.add` in push insert the element at the beginning of the list and shifts element at 0 position and subsequent elements one position to the right. `.remove` has the same issue above. Therefore both of these two methods have shifting operations that take O(n) time.

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 we use add 5 instead of double, we will need to resize much more frequently (resize every 5 elements). Each resize needs to copy all existing elements into a new array which takes O(n) time.