# Stack Data Structure Time Complexity Analysis

Bryant Passage

**Introduction:**

In this project, three stack data structures were constructed with the goal of calculating the time complexities of each one. A stack is an abstract data type which has the concept of "last in first out" when dealing with elements entering and leaving the stack. As the stack increases in size, its memory allocation needs to be expanded every time the stack starts to become full. Here, we implement three types of stack data structures: a constant growing stack, a double growing stack, and a Linked List stack. Each of these data structures have their own advantages discussed in the theoretical analysis of the section. The constant growing stack and the double growing stack were implemented with standard C++ arrays (not vectors) created on the heap. The linked list stack used nodes that were created in the heap and connected with pointers. Each stack data structure was pushed with 10,000,000 integers and timed after every 1,000 pushes. With these three stack data structures, we can find which one has the fastest asymptotic running time and which one would be best suited for different cases in programming.

**Theoretical Analysis:**

A push operation for all three stack data structures normally takes *O(1)* times. This is assuming that the that the two array-based stacks do not need to grow in size and the linked list only needs to add a new node at the end of the list and change the tail's next pointer to the new node and also change the tail pointer to the new node as well. All these push operations require a constant number of operations in which we can simplify to *O(1)* since constants do not matter in big-oh notation. However, if the two array-based stacks DO need to grow in size, their asymptotic time complexities differ. For a constant growing array stack, if the stack needs to grow every time a push is initiated, then the time complexity for such an operation is worst case $O(n^2)$. This is because every time a push operation occurred, the array would need to be copied to an array with a size that is bigger by one and then the new element would be pushed into the stack. This would result in a total operation of (1+2+3+4+…+(n-1)+n) for *n* pushes which results in a $O(n^2)$ for worst case. However, the average time complexity for this constant growing stack implementation is *O(n)*. This average is taking to account the worst case scenario and the best case scenario where the stack grows by a bigger constant rather than just one. The bigger the constant growing size, the closer the average would be towards the best case scenario of *O(1)*.

For a double growing stack data structure, the best case scenario is the same as the constant growing array and the linked list which is *O(1)*. The worst case scenario for this array is *O(n)*. This is because, even though the first time the array doubles and copies over the elements from one array to another, the average of these operations would result only in a constant number of operations over an *n* amount of inputs which would be *O(n)*. The average/amortized case for this data structure however is *O(1)*. This is because the fact that the array is doubled, so the array would grow twice as big every time the array is full, allowing for more push operations before having to duplicate and copy over the entire array to another array.

For a linked list, the best, worst, and average case for a push operation is *O(1)*. One push operation takes a constant number of operation and the linked list does not need to duplicate and grow after a certain amount of push operations which means that the constant number of operations for *n* inputs would still be *O(1)*.

**Experimental Setup:**

A. Machine Specifications:
    - Custom built PC “
        - Ryzen 7 3700X @ 3.6GHz Base → 4.4Ghz Boost*
        - ASUS X470-F Gaming Motherboard
        - 16 GB Trident Z RGB 3200MHz DDR4 RAM
        - Nvidia GeForce GTX 1070 Founders Edition

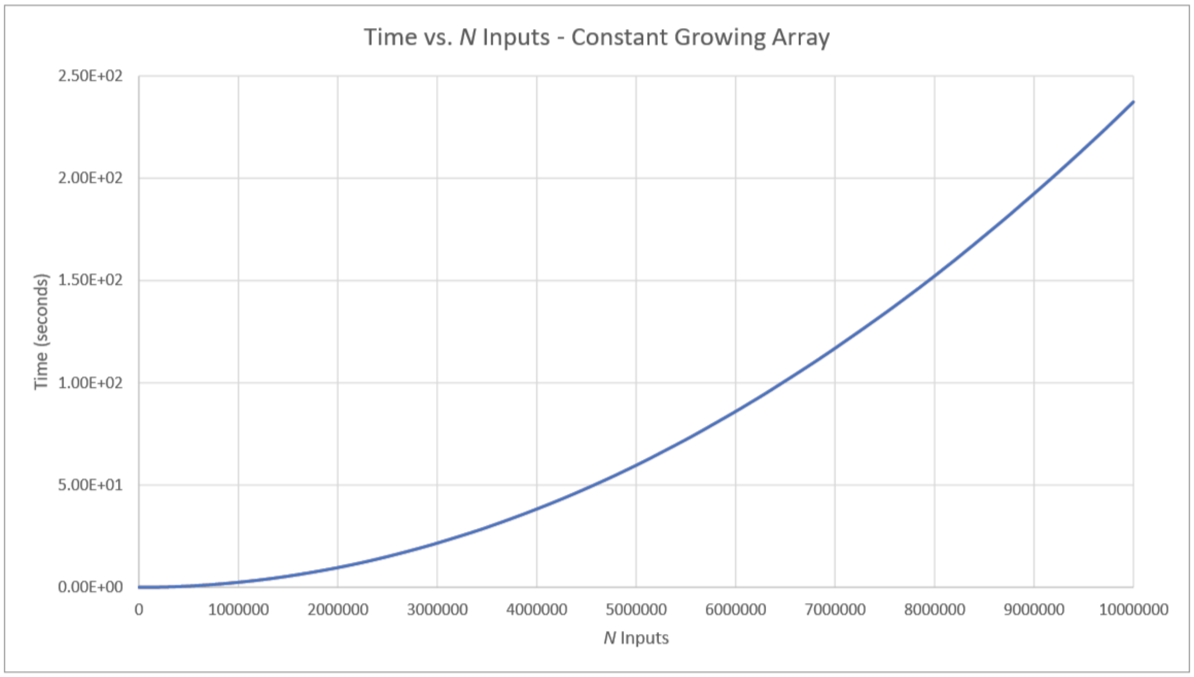
*Note: The boost specifications of this CPU vary throughout the time the program runs. Therefore, the comparison should only be made relative to the other tests conducted with the same test bench.

B. Test Setup:
    - Each stack data structure was tested with 10,000,000 push operations of the integer datatype. This was to ensure that boost functions of the CPU did not fluctuate the overall push operations over time.
    - Constant growing array stack
        - Trial 1:
            - Initial Size = 100
            - Constant Growth = 1,000
        - Trial 2:
            - Initial Size = 1,000
            - Constant Growth = 1,000
        - Trial 3:
            - Initial Size = 1,000
            - Constant Growth = 10,000
    - Double growing array stack
        - Trial 1:
            - Initial Size = 100
        - Trial 2:
            - Initial Size = 1,000
        - Trial 3:
            - Initial Size = 10,000
    - Linked List
        - 2 Trials conducted to validate results

**Experimental Results:**

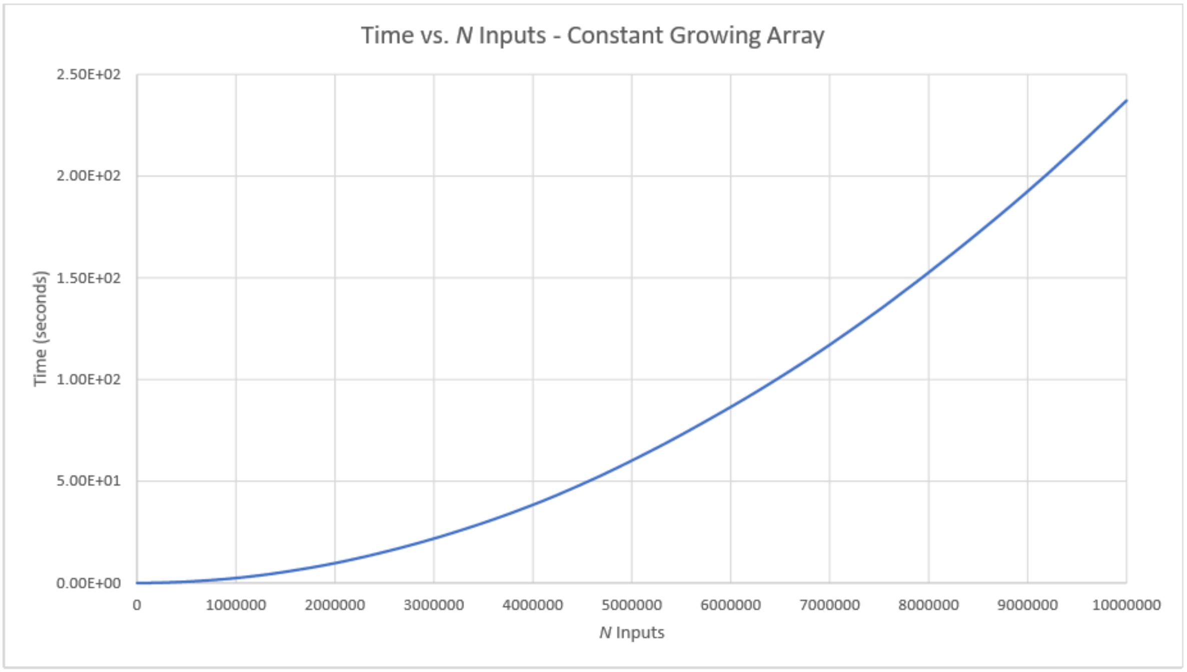
Constant Growing Stack:

- Initial Size = 100
- Constant Growth = 1,000



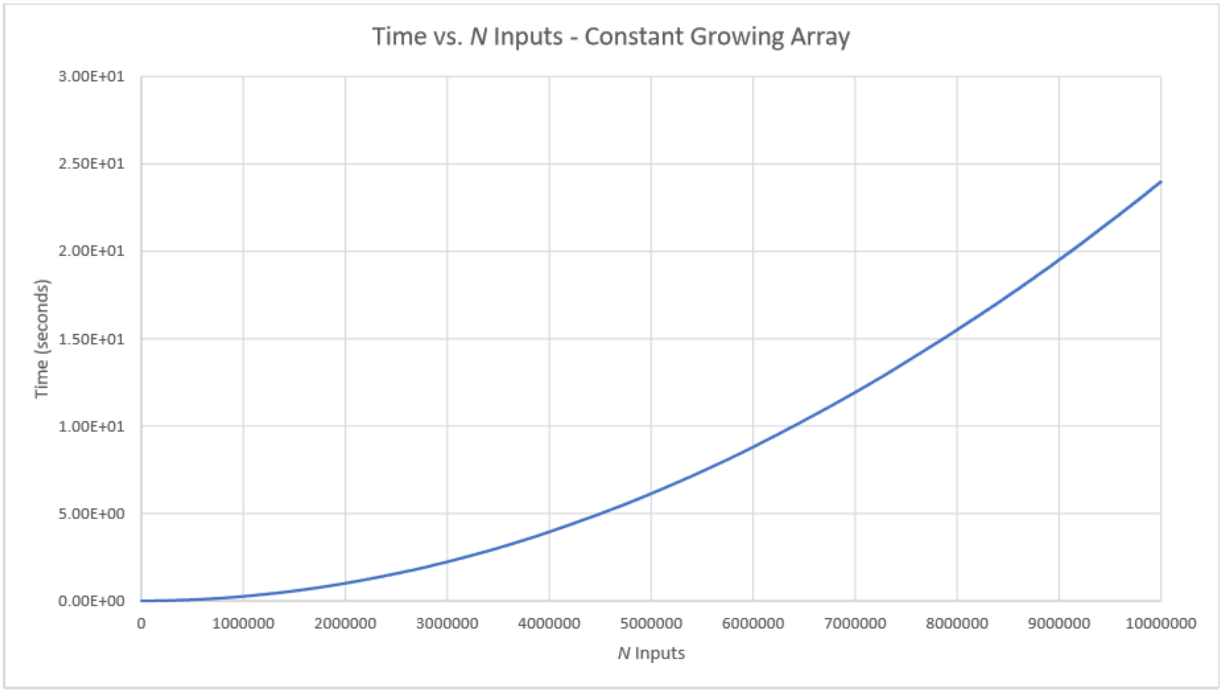
Constant Growing Stack:

- Initial Size = 1,000
- Constant Growth = 1,000



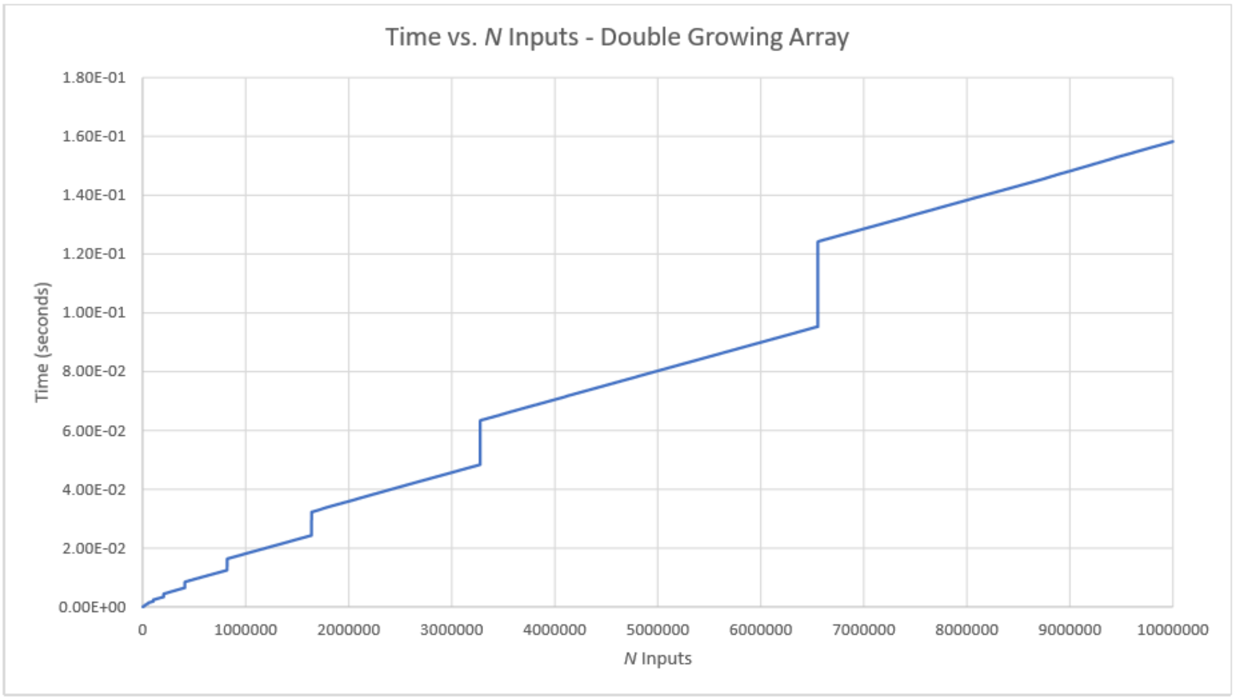
Constant Growing Stack:

- Initial Size = 1,000
- Constant Growth = 10,000



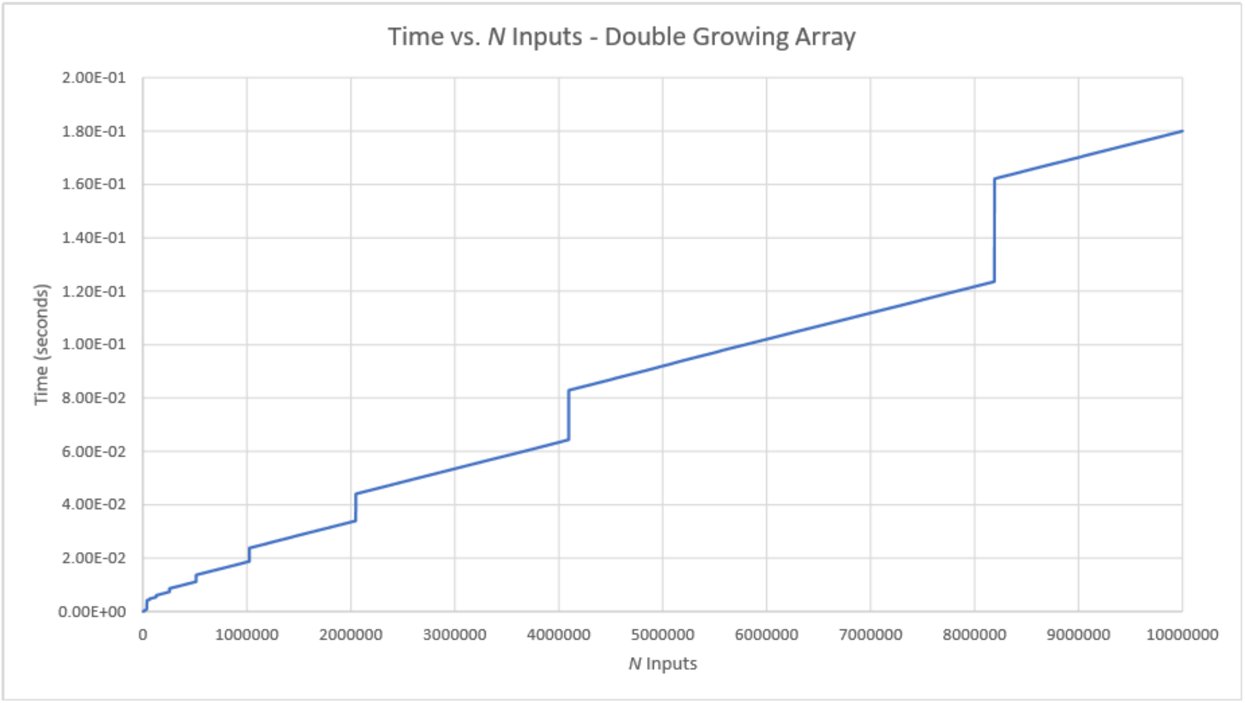
Double Growing Stack:

- Initial Size = 100



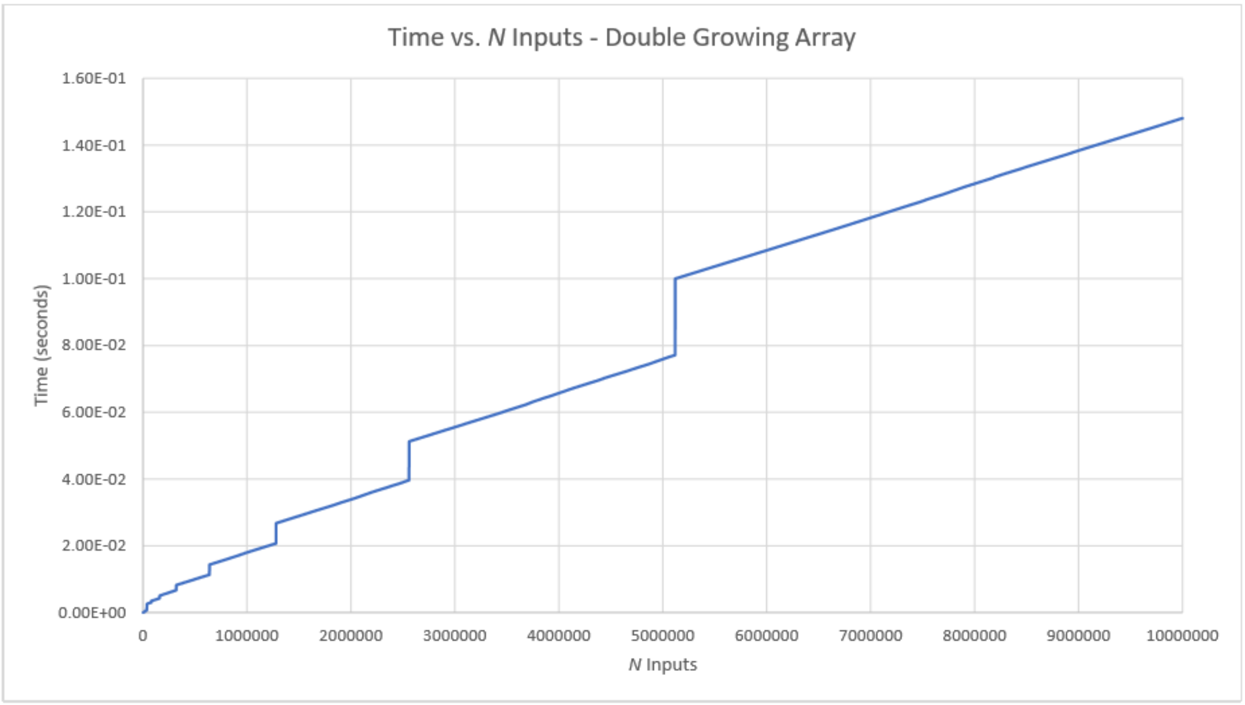
Double Growing Stack:

- Initial Size = 1,000

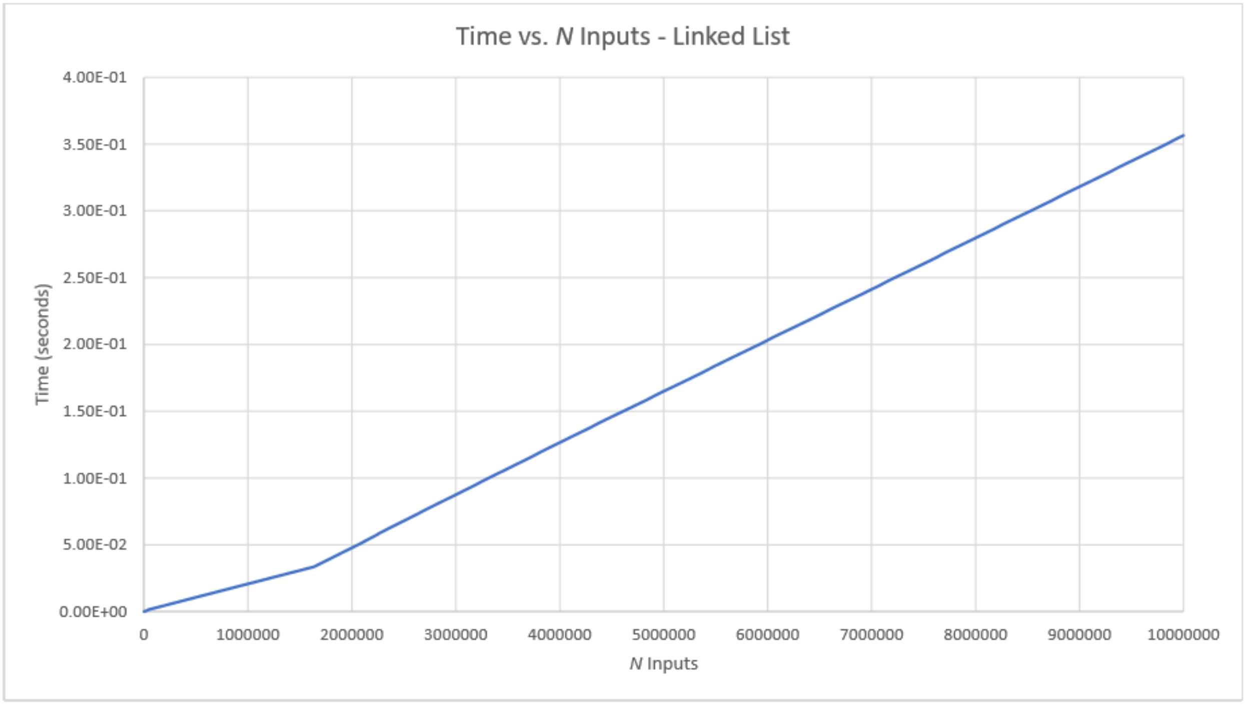


Double Growing Stack:

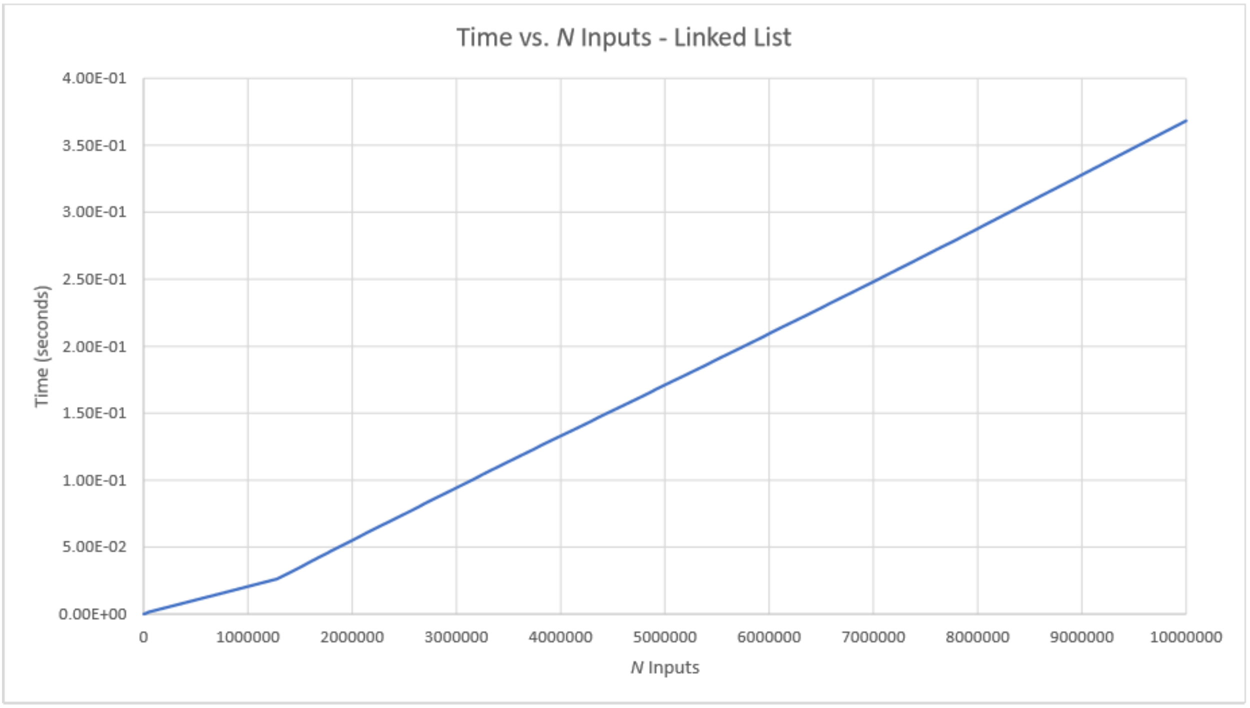
- Initial Size = 10,000



Linked List 1:



Linked List 2:



**Conclusion:**

In terms of raw time, the double growing array-based stack was the best performing out of the three. This follows the theoretical analysis that the doubling stack has an amortized cost of *O(1)* because of the fact that the array does not need to duplicate the array as much over time. However, we also discovered that the linked list also has a push operation cost of *O(1)* as well. However, big-oh notation hides constants that are actually important when comparing two algorithms that have the same time complexity. In this case, a linked list implemented stack datatype takes longer to complete 10,000,000 push operations because you must also initialize 10,000,000 nodes before linking them together in a chain, and creating each node takes longer than initializing an integer and pushing them into a C++ array. The constant growing array did the worst out of the three due to the fact that it requires frequent duplicating of the array after a constant number of push operations. However, the advantage of using the constant growing array stack is the predictable usage of memory. Since the array grows at a constant rate, the user would know how much memory is being allocated to the stack after each push operation. With the linked list and double growing stack, the memory usage is much higher as more memory needs to be allocated for a node and double the memory is allocated whenever the double growing stack needs to increase in size.

Overall, the theoretical analysis done for each of the three stack data structures were very accurate. With an amortized push cost of *O(1)*, the double growing array stack performs the best in terms of time to complete 10,000,000 push operations. Linked list comes next due to its same cost of *O(1)* push cost, however, initializing nodes and rearranging pointers takes more time than just adding a new element to an array. Lastly, the constant growing array stack has a big-oh time complexity of *O(n)* on average and can be seen during the test push operations done in this experiment. The experiment matches the theoretical analysis for the time complexities of the three stack data structures: the constant growing array-based stack, the double growing array-based stack, and the linked list.