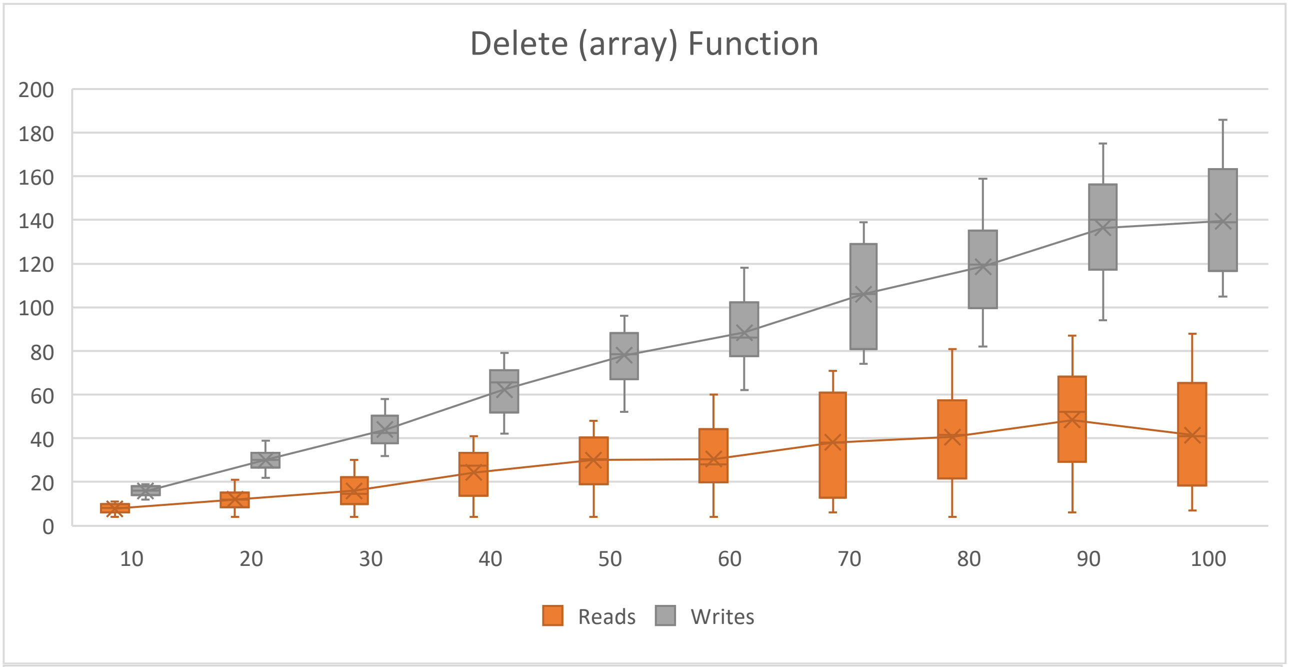
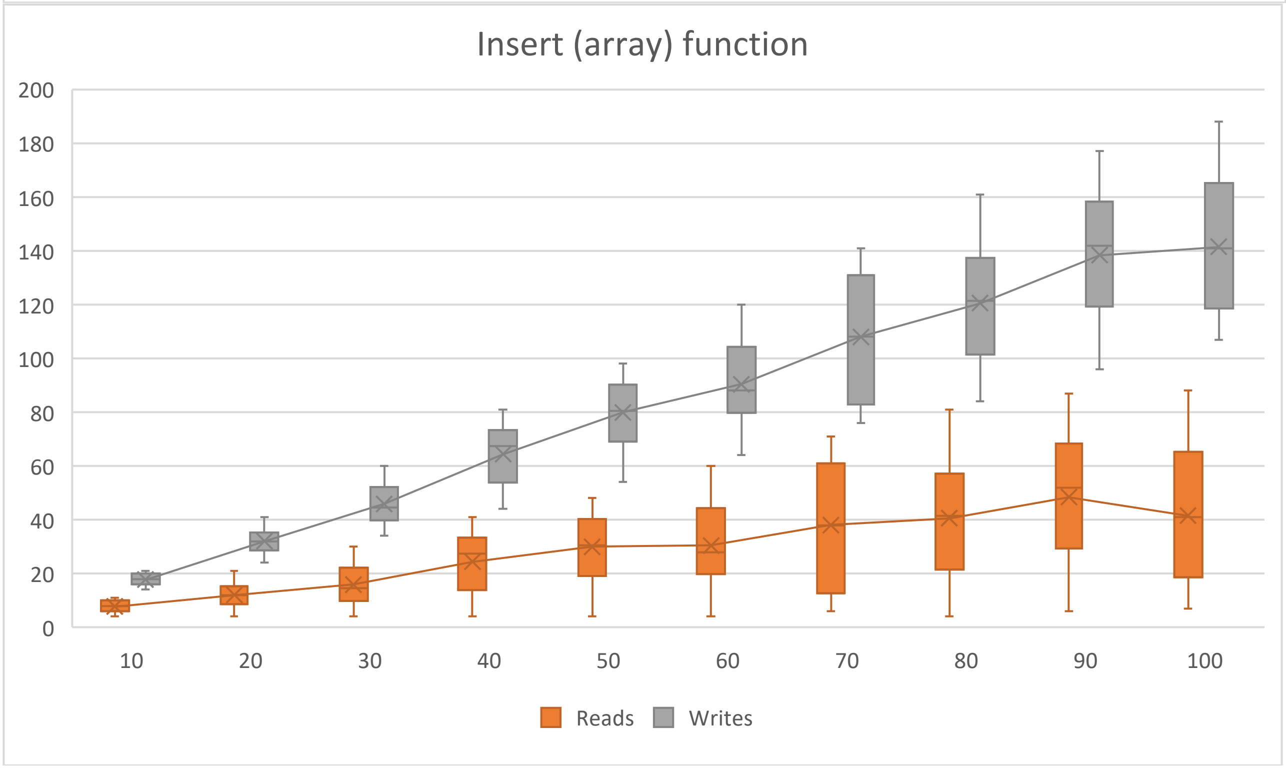
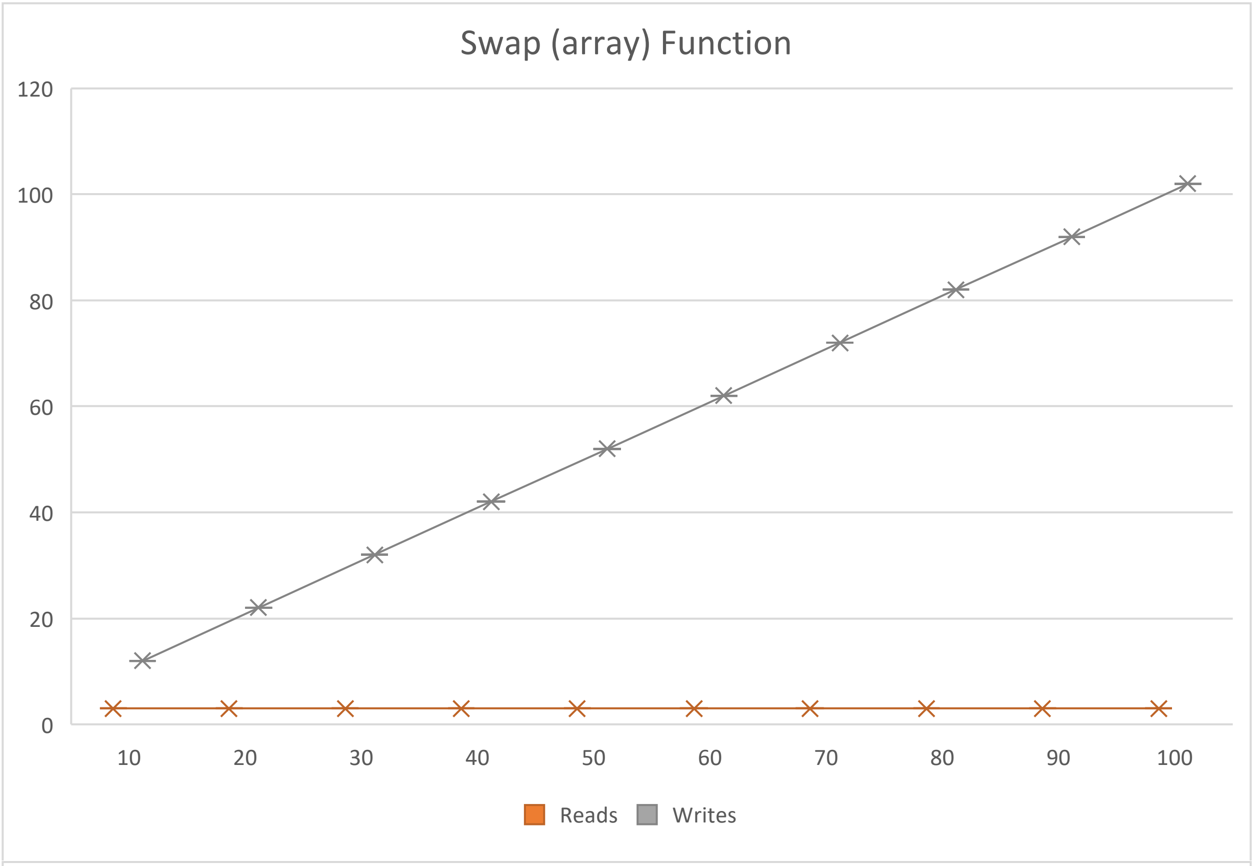
Habiba Mostafa
1070307
mostafah@uoguelph.ca

Part B – Analysis

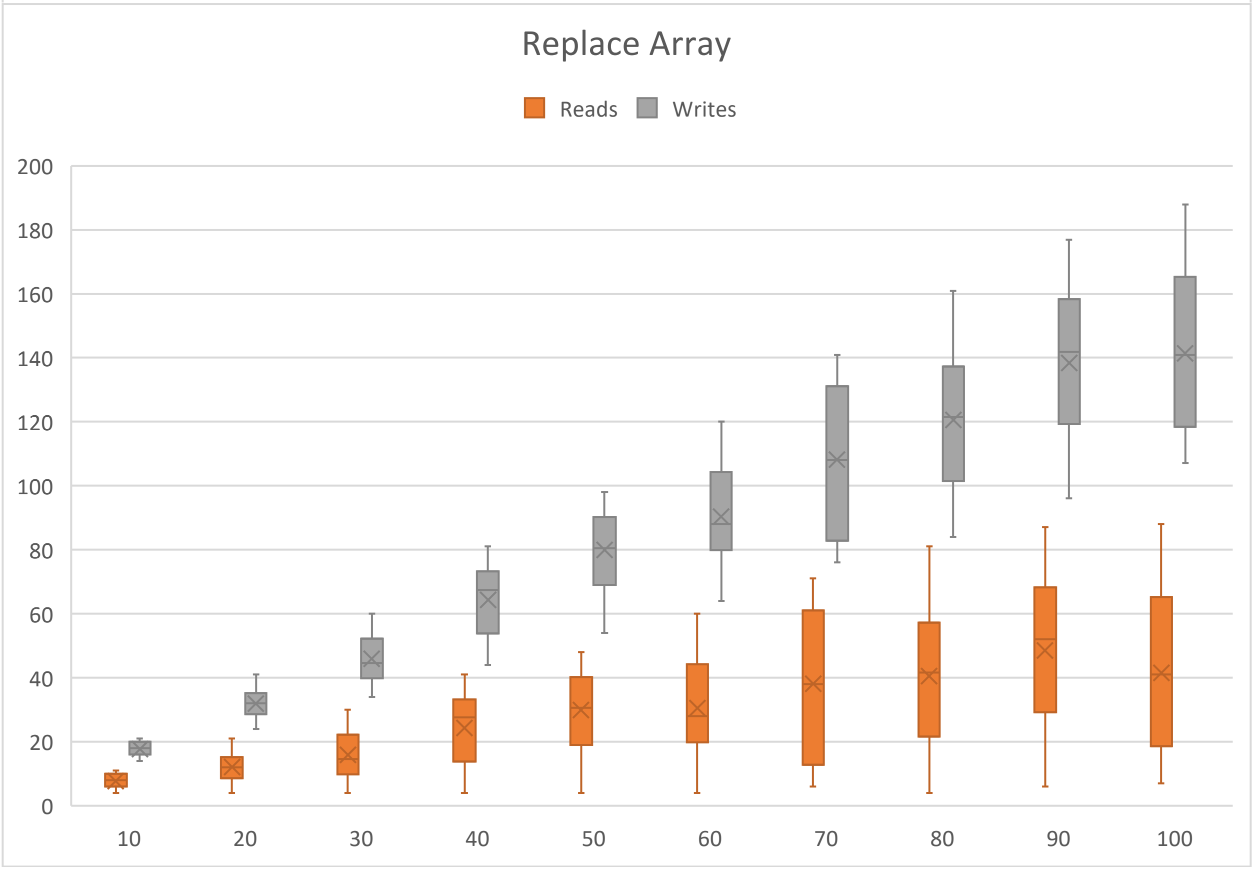
Graphs



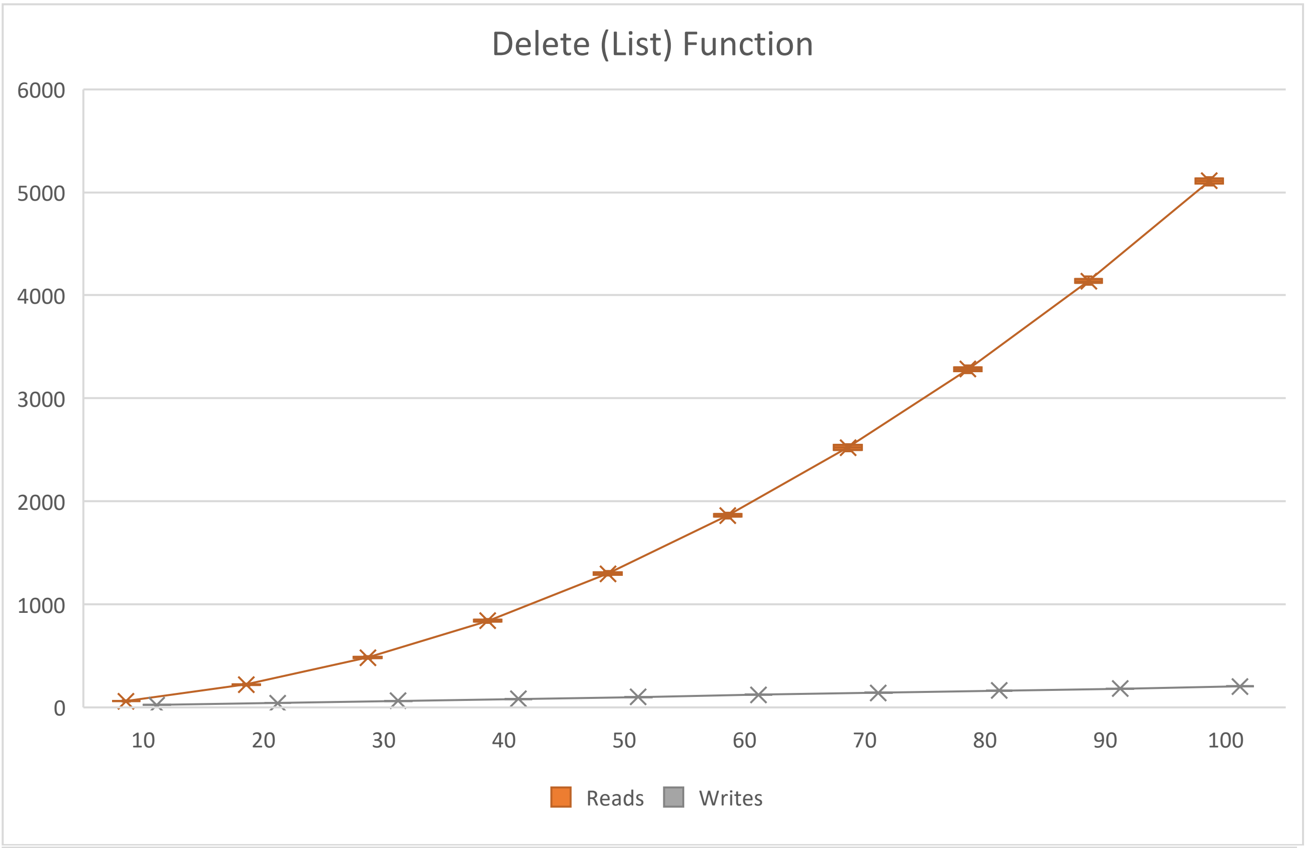


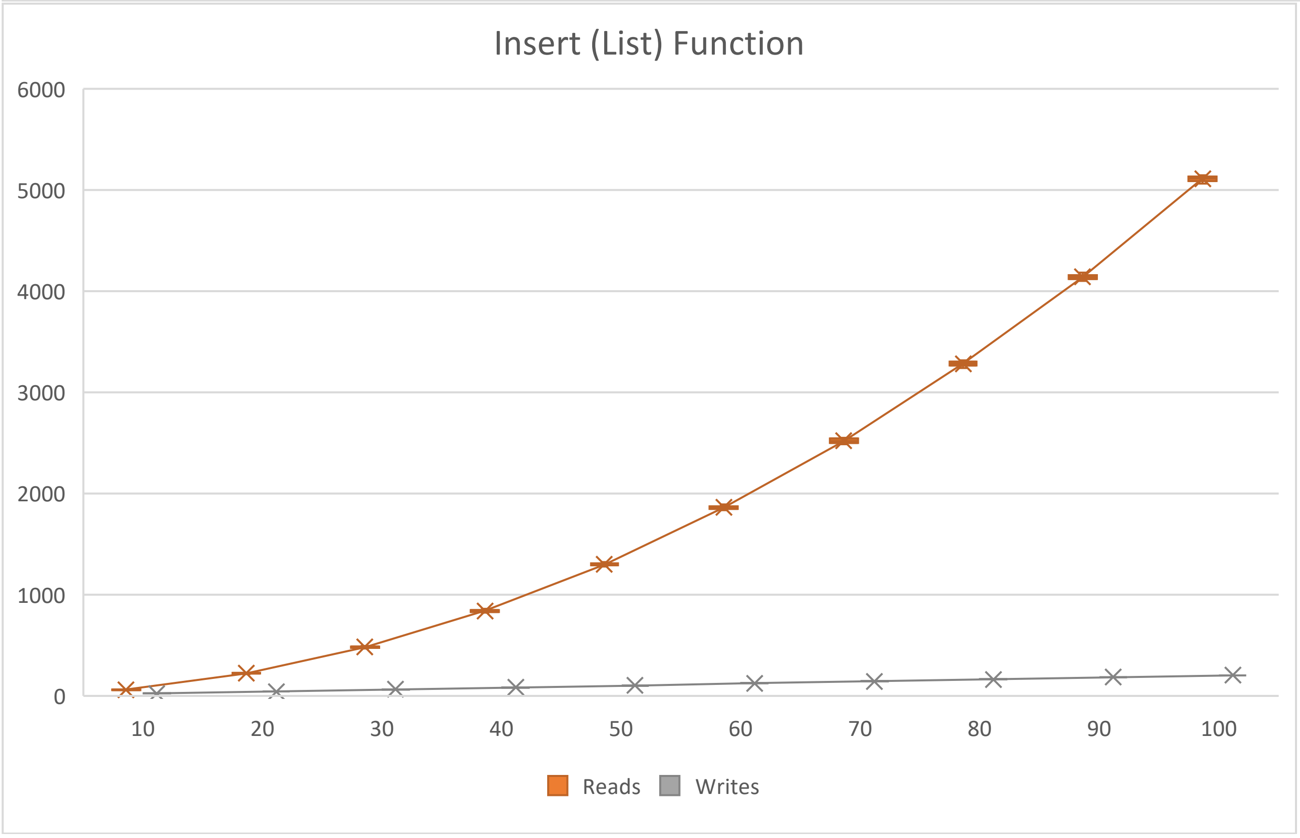


Swap (array) Function
120
100
80
60
40
20
0
10
20
30
40
50
60
70
80
90
100
Reads
Writes



Replace Array
Reads
Writes
200
180
160
140
120
100
80
60
40
20
0
10
20
30
40
50
60
70
80
90
100


Delete (List) Function
6000
5000
4000
3000
2000
1000
0
10
20
30
40
50
60
70
80
90
100
🟧 Reads
⬜ Writes


Insert (List) Function
6000
5000
4000
3000
2000
1000
0
10
20
30
40
50
60
70
80
90
100
🟧 Reads
⬜ Writes

Discussion

While arrays and linked lists differ in many aspects, one data structure is usually more superior than the other, depending on the algorithm that is to be implemented.

In the case of retrieving data, for example, an array is likely to be a more efficient data structure, given the ability of accessing an array at a given index, using notation as simple as array[index]. Navigating through an array to find a specific element at a known index location is a quick, shorter process as opposed to a linked list. In terms of runtime, using an array to access a known index is an efficient process. A linked list would require traversing from the head and working through until the specified element is located. Since random access is not permitted, sequential access starting at the first node will be performed, hence runtime in this scenario would vary depending on how far into the linked list the element is.

Practical examples similar to this one where an array would be superior include knowing the array size, as it saves memory. Rather than allocating memory for each node and its pointer in a linked list, an array requires the allocation of space per element, which makes it more efficient in terms of memory allocation. Some might even argue that accessing array elements will always be faster compared to a linked list, given that array elements are stored in contiguous memory and are easily cached, while nodes in a linked list can often be located in random memory locations that require time to track down.

On the other hand, in the case of adding data or removing data from an existing data structure, the use of a linked list is highly efficient. The ability to traverse a linked list as needed to perform the required action of inserting a new node at a given location is made simple in a linked list, as opposed to an array. It requires less runtime, especially if a new element is to be added to the beginning of the list. A new node would simply be added to the list and positioned before the head, becoming the new head of the linked list. An additional pointer now links this new element to the existing first element of the list. No elements are required to shift over, unlike in an array, which increases runtime and reduces efficiency. This applied to deleting nodes from a linked list as well. Another factor to be considered in this case is memory. Memory is allocated within linked lists during runtime or execution, while memory in the case of arrays is assigned at compile time. This makes dynamic allocation smoother in linked lists. This was roughly tested through the use of ds_delete and ds_insert in the array management system and similarly for the linked list management system.

Practical examples similar to the ones above - in favor of linked lists - as demonstrated through this assignment's tests include the need for constant time. We deal with linear time when using linked lists which provides accurate data to serve for time predictability and real-time computing purposes. Upon testing and printing run time, I could usually tell that linked lists are more consistent in terms of runtime. For research and data sciences purposes, linked lists are likely to be more concise.