Johann Jaramillo

CSC311-02: Data Structures and Algorithms

May. 17, 2024

Report on Data Structures

1. **Introduction**

In our Data Structures and Algorithms class, we have taken a look at eight different implementations of four unique abstract data types. The abstract data types (ADTs) we implemented are: List, Stack, Queue, Set, Map. In this report, we will take a look at the time complexity of each individual method involved in the implementations of the ADTs. Furthermore, we will attempt to optimize any methods possible and report on whether our improvements changed the time complexity of the given data structure. Additionally, the data structures will be updated to handle any types of data types, not just “String” or “int”. Lastly, the paper will summarize our findings concisely and determine whether or not the changes are effective.

***Key Terms: Time complexity, List, Stack, Queue, Set, Map, ADT, Linked, Array, Hash, Nodes.***

To begin, let us briefly explain each ADT, as well as the implementations of each.

*List*

The interface List is used to insert, sort, and access data at certain indexes. The implementations of this interface are LinkedList and ArrayList. LinkedList chooses to store data via Nodes, which contain only its own data and the next Node to be accessed. A LinkedList must therefore have a head node and a tail node, and you must iterate through each node before the one you attempt to access. An ArrayList chooses to store data using the data structure array, giving indexes to the data you are storing as opposed to nodes like a LinkedList. ArrayList also has the capability to increase length if necessary, a common trait of implementations.

*Stack*

Stacks are similar to Lists, with the restricted functionality to only be able to operate on the most recently added data. The implementation of Stacks are LinkedStack and ArrayStacks. As we can see, the “Linked” term means that the data structure will be using nodes, and the “Array” term means it will be using arrays. These implementations follow the same rules as ArrayList and LinkedList, with the difference being that you can only operate on a last in first out order, like a *stack* of plates.

*Queue*

Queues follow the same functionality as Lists, but data that has been stored the longest is given priority to be used next. This is a first in first out order, like a line at the store. Once again, LinkedQueues and ArrayQueues use their respective object types to store data with nodes and arrays respectively and conform to the Queues’ guidelines.

*Set*

Sets follow functionality of Lists in that you can access data via an index. However, sets do not allow for duplicate elements. Our implementation of set is HashSet, which is very different from previous implementations. HashSet takes in data, generates a unique hash code for it, applies a mathematical formula that is typically hashCode%lengthOfSet, and places the data into an array of nodes known as a hashBucket, declaring the data as head node if one is not declared yet and attaching it as a tail if it does exist. Furthermore, HashSets are given a load factor, which determines a HashSet will need to resize to double length when the hashSet size meets load factor \* length. When resizing is done, the previous elements are dispersed throughout the new length by the same formula of hashCode%lengthOfSet.

*Map*

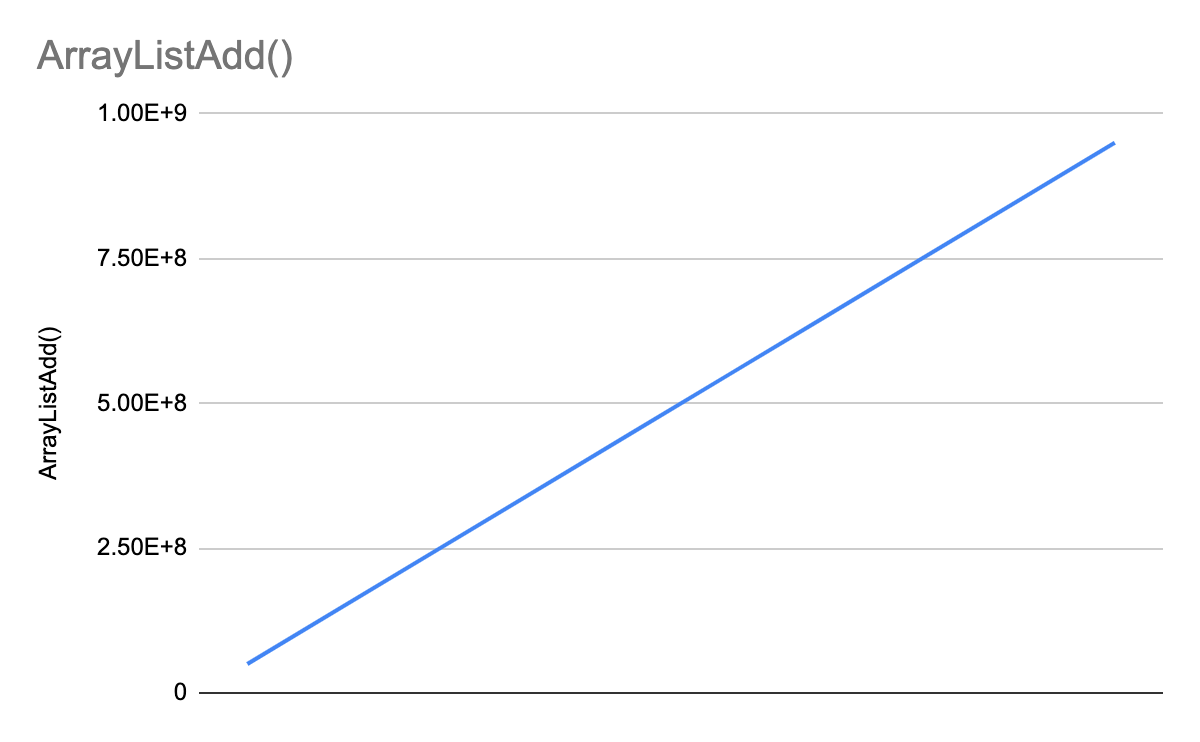
Maps stores data into special types of nodes called MapNodes. These Nodes are the same as regular nodes, only they also store a key with the given data. Furthermore, there should be no duplicate keys in any MapNodes, with the data of the previous key being overwritten if you attempt to insert a duplicate key. An implementation of a Map is HashMap, which is similar to HashSets in that data is placed into hashBuckets as MapNodes. It follows the same procedures and processes as HashSets, only it checks for duplicate keys rather than duplicate data.

1. **Report**

Each of these data structures will be tested against of 50 sets of randomly occurring int values, with the sample ranges starting at 1000 and increasing by 1000 for every additional test up to 50,000. I’ll analyze the time complexity and collect data from every test, comparing the differences and choosing to improve those that perform worse. Then, the tests will be ran again and collected again so as to demonstrate the improvements to the data structures. The data tables and charts will graphically display and prove the changes in our structures.

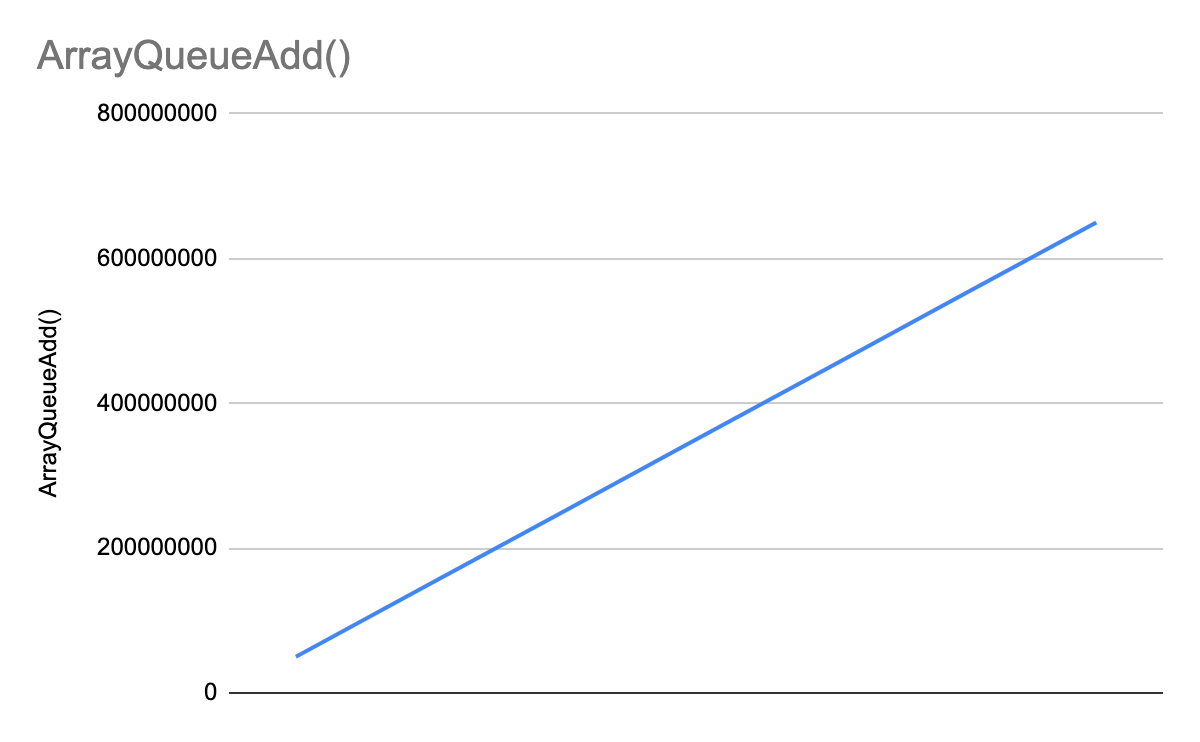
*ArrayList:*

My add() Method maintains O(n) complexity most of the time, but if the arraylist is resized to fit more values then the complexity changes to O(n\*n) if resizing occurs. When resizing does not occur, add() loops as many times as however much data you add, i.e., a 1:1 ratio. However, when resizing occurs, it essentially squares your loop cycles. When comparing adding 2000 ints to 3000 ints, the difference of cycles is 1 million which is 10002. This difference carries on until infinity, but begins somewhere after resizing past 1000 ints as the difference is less than n2 in the first case.



We determine that there must be improvements to be made, but this complexity is not terrible. Insert(ind, obj) has O(1) complexity, as the loop counts will never exceed the size of our data structure since we know where insertion will occur and no resizing will take place. We can assume that all other functions will take less loopCycles than add(), as they do not have nested for loops and conclude that apart from add() and consequently resize(), no other methods should be revised.

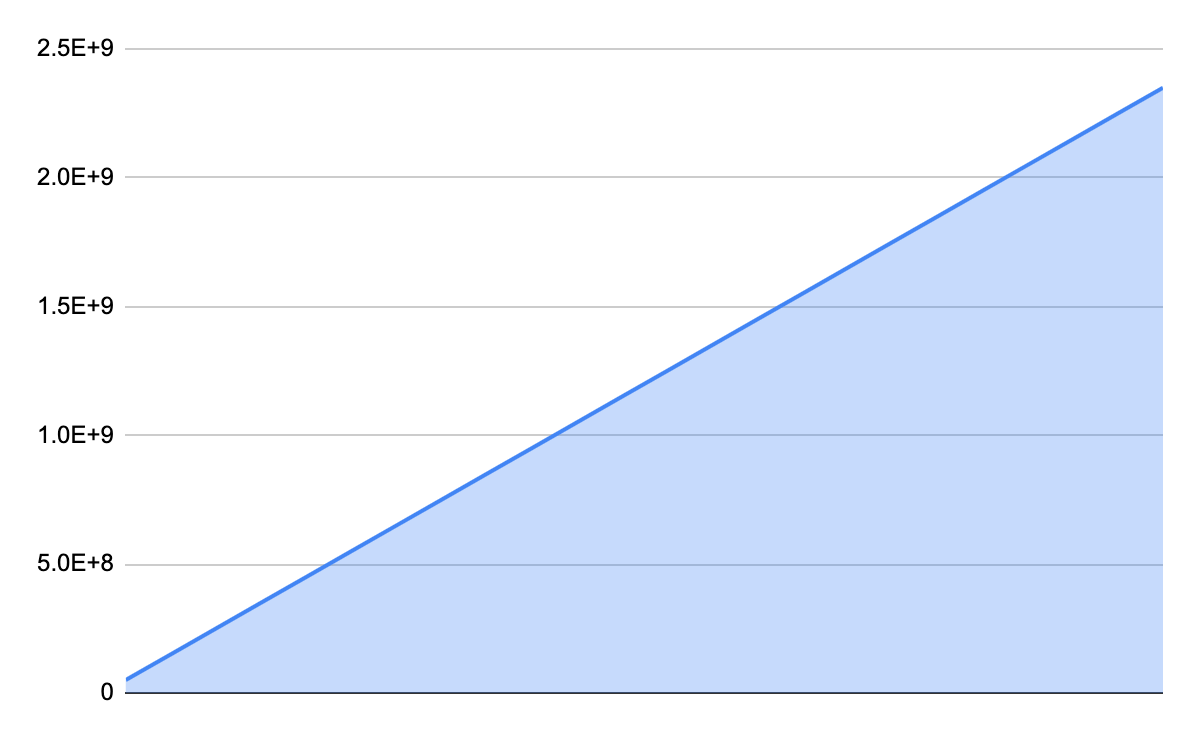
*ArrayQueue:*

This data structure is nearly identical to ArrayList, only we cannot insert nor remove by indexes and only via peek(), pop(), and add()/push() to either the tops or bottoms of the queue. 

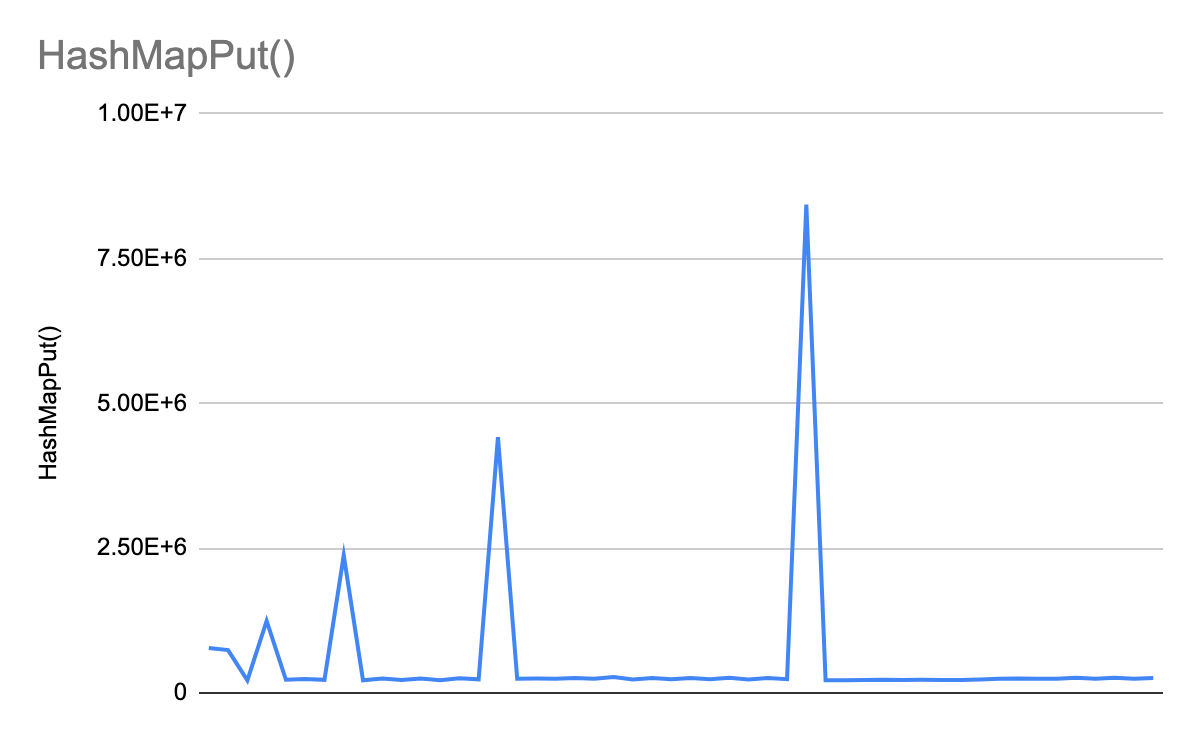
As we can see, the complexity for ArrayQueue add() remains the same to ArrayList, and therefore we infer that it shares the drawbacks of resize() and should be revised.

*ArrayStack*

Again, this data structure is identical to ArrayQueue with the exception of data handling; This structure only allows for operation of data on top of the stack via push(), which will also call resize() if the array is not large enough to accept new data. We can see the exact same pattern as the previously mentioned and no other methods contain loops or nested statements, therefore, only push() and resize() must be revised.



*HashMap()*

HashMap allows us to store data by inserting it into nodes that are then stored into buckets. This is handy because we can access data from large quantities with less operations if we know where the data is stored. put() places data into a node, which is placed in a bucket, and checks for collision. If the bucket has no room, it is resized by its load factor, and this is apparent in our data set: 

Initially, there are more loops required because we are incrementing our sets by 1000 and our buckets initial capacity is only 16, so quite a bit of resizing occurs. However, resizing is not as much of a problem towards the end of the set at 50,000 ints and loop count stays close to 250,000 cycles regardless of the amount of data. Only when rehashing occurs does the cycle count increase dramatically, leading me to believe that my put() has O(log n) complexity, but the rehash() method has O (2n) complexity as it will require double the amount of cycles as the previous rehash() call required, seen in the chart. Rehash should likely be redone. Next, we have get()

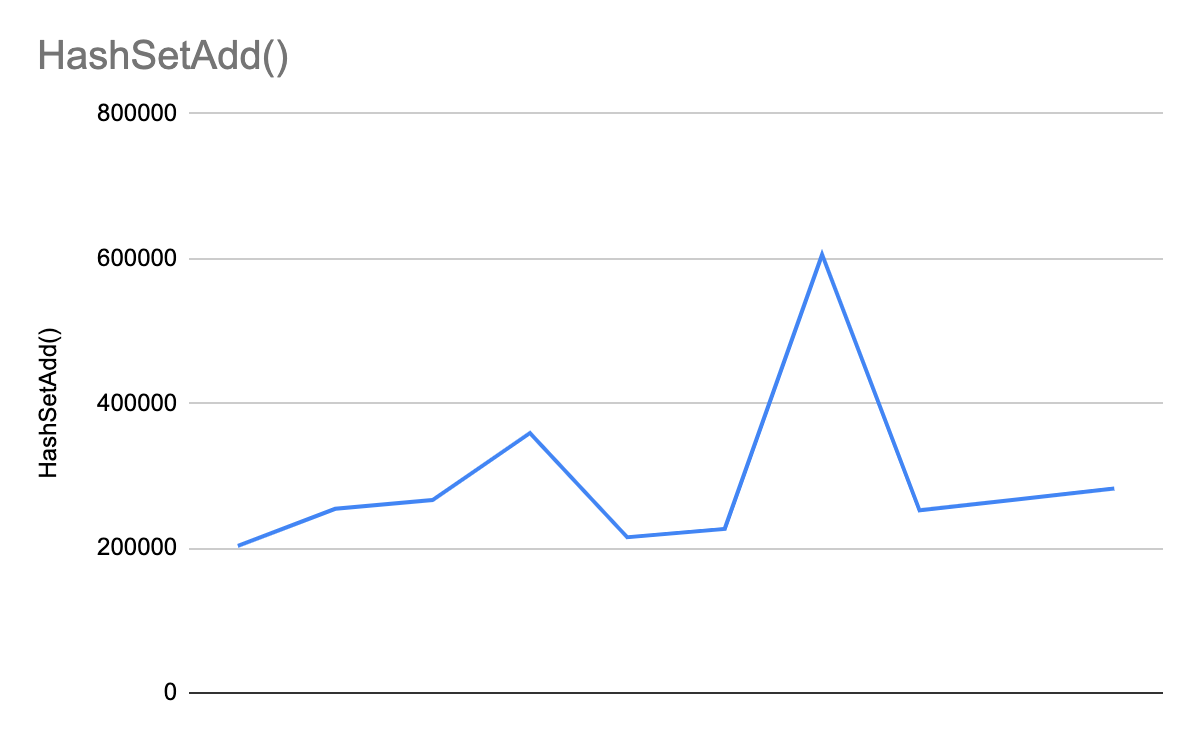


Which searches our Map and retrieves the value for the key given to it. Generally speaking, it found the smaller values sooner but overall cycles did not deviate more than 50,000 despite the size of our HashMap being 2,500,000 by the end of put(). This tells us that get() must have O(log n) complexity, as its cycles are not greatly dependent on the input but rather the Map size itself.

containsValue() actually could not go through the entire data set, which tells me it is either not fully functional or greatly under optimized and should use a search algorithm to improve, as the very first call took nearly the full size. The same can be said for containsKey(). All other methods take in no input, and therefore are not dependent on the input but the Map itself.

*HashSet()*

Data is stored into nodes, which are placed into hashBuckets. Data is compared to existing data and prevents addition of duplicates, and hash Buckets are resized if the load factor is exceeded.



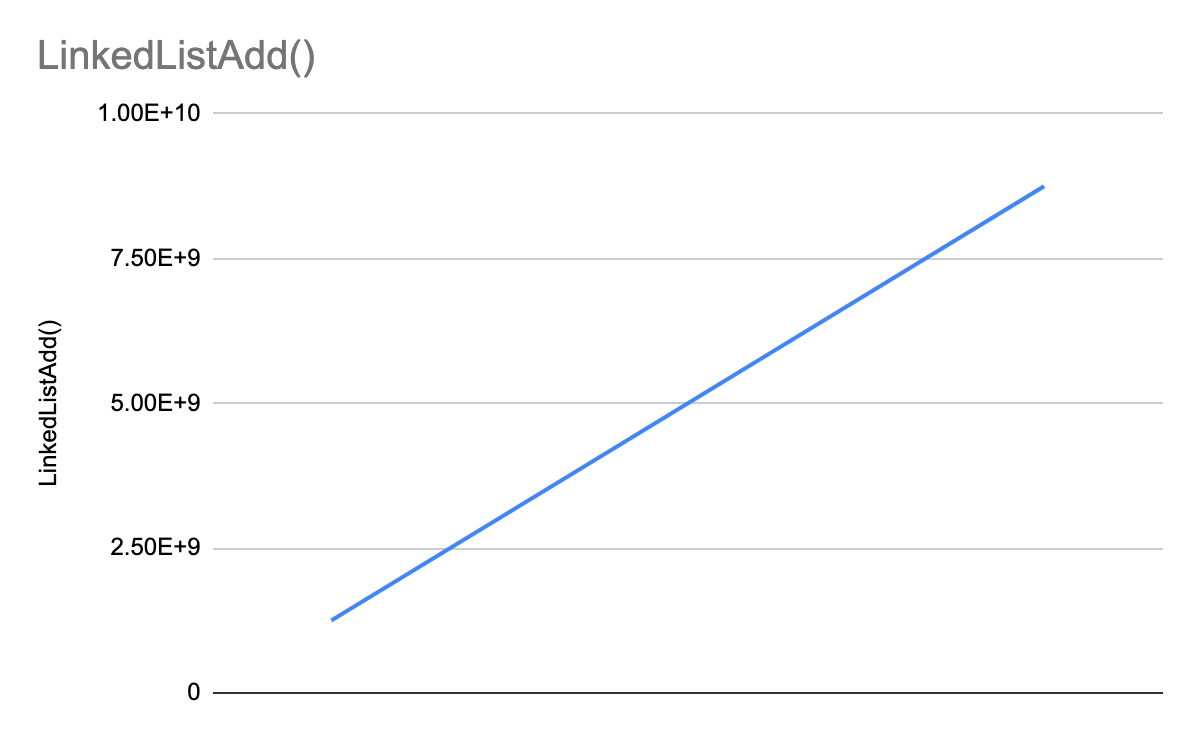
HashSet() add() does well in adding during small data sets(50), as cycles generally were 3 when adding data, with a bit more colliding and a lot more when rehashing. However, when working with the large set that goes from 1000-50,000, cycles ranged from 200k to 300k based on collisions, with rehashing taking more and at least 1.5x the previous rehash cycle count. After rehashing occurs, cycle counts drop to below the previous non-rehash count, as seen in the chart. My add thus has best case complexity of O(1) and worst case scenario of O(n). Going by this, we determine rehash() should be rewritten, although it is useful in keeping add()’s cycles low. size() iterates 1 by 1 through the set.

*LinkedQueue*

LinkedQueues store data in nodes, and the node is appended to become the new head of the queue. Only the Add() function takes in input, and when testing with a small set the cycle count took 1:1, however this count greatly increased to integer overflow counts due to findTail() method iterating 1 by 1 through the queue. Unfortunately, this means I could not gather accurate data for larger sets but it does mean that my best case add() is O(n) and worst case is when findTail() grows exponentially, likely O(2^n) and should consider immediate replacement.

*LinkedList*

Stores data into a list of nodes.



My linked list add() takes much too long, growing by a rate of 2.5 billion per 1000 additional items to add… this function nearly crashed my computer, largely in part due to findTail(), which is called every single time. If we optimize findTail() and apply a conditional workaround to adding nodes without calling findTail(), the linkedList can become something much greater. For now, since my computer cannot handle adding large data sets through linkedList, I can’t test any other functions.

*LinkedQueue*

Same functionality as LinkedList, only we add nodes to the end of the queue and process nodes at the front of the queue as rules for data handling.

Add() takes O(n) complexity, taking the same amount of loops as LinkedList per 1000 items. You would think that as a queue, it would be more efficient but as we have to findTail() again to append the new addition, we’d see great improvement’s if we could access the tail without iteration of 1 by 1 every time.

*LinkedStack*

See LinkedQueue.

1. **Conclusion**

**Not all data structures are created equal.**

Between Lists, Nodes, and Maps, I found that for my code at least, Maps (Specifically HashMaps) came out to be much more efficient in terms of adding and accessing data when compared to lists and then our implementation of linked nodes.