Empirical Analysis on Arrary List versus Linked List: Java

Paul Kevin Short

***Abstract*— This paper addresses an empirical analysis on the efficiency of Java's implementation of an Array List and a Linked List. The tests include running the methods: add (or insertion), get, and remove while the List grows up to a size of 200,000 and logging how fast these methods will run. This research is intended to resolve questions that a software developer would have about which data structure will perform the best, and what size will bring these data structures to their knees.**

I. INTRODUCTION

A major question that is raised whenever the discussion of whether one should use an Array List or a Linked List to store their data is how does their actual performance compare to their asymptotic complexity. It is known that when using a Doubly-Linked List, which is the Linked List implemented in the *java.util* library, building a fairly large Linked List shouldn't take anytime at all asymptotically speaking because adding to a Linked List is done in *O(1)* time. Comparing this to the asymptotic complexity of an Array List, worst case, it will have to copy the entire array if it had to add a value to the front. This makes its complexity *O(n)*, which is slower than Linked List's add method. The difference between the add methods actually lies within knowing where to add the value, because in an Array List the object knows where each element is and can access it in *O(1)* time, whereas within a Linked List the object will have to traverse to the index that's being added to.

The test code will be tested on two different operating systems, Linux OS on a cloud server and a local machine running Mac OS. The Mac OS is running on a Intel Core i7 at 2.2 GHz on 16 GB 1600 MHz DDR3 of RAM. The Linux machine is running on (find out from John). This is to test whether there is a major difference between how each operating system stores, manipulates, and cleans up the data the Java application is handling.

II. PROCEDURE OF TESTING

A. Sizing

In order to see how the different list implementations run when used in software that requires heavy lifting, a starting size of 10,000 was chosen as well as incrementing the size by 10,000. This will not only show which data structure could handle large data sets the quickest, but it may also provide a size threshold for these data structure in terms of how much data can in handle in a reasonable amount of time. The threshold will be initially set to 200,000 to see if the data structures become too slow once the threshold is reached.

B. Insertion Method

The decided procedure for testing the insertion method of the two data structures was to first, initialize an empty list of both data structures and independently time how long it will take each List to grow their size to 200,000 in increments of 10,000. This is resolving how fast each data structure can insert 10,000 randomly generated integers between 0-1000 at a randomly generated index.

C. Get Method

A similar procedure was used for testing the get method of the data structures which starts at initializing the list to be empty and then growing it in increments of 10,000. Since an Array List can access memory in *O(1)* time because the memory is in sequence, it was decided that testing the get functionality would have to involve a large amount of retrievals. To mimic heavy lifting software, the number of memory accesses will be set to 1,000. This entails that at each size tested each get method will be run 1,000 times and timed to see how the runtime performance is.

D. Remove Method

To test the *remove()* method of each data structure it was decided that the method would be called 10,000 times to remove a single item at a random index of the list. This will show how each data structure can handle large amounts of data removals in a reasonable amount of time. Since 10,000 items will be removed each list will be initialized at 200,000 and timed 20 times (increments of 10,000).

III. ADD METHOD RESULTS

As mentioned in the introduction the test code was run on two different operating systems and timed in milliseconds.

A. Mac OS

As shown in Figure 1 the performance of Linked List vs the performance of Array List differs from the asymptotic analysis such as, when both lists initially add 10,000 integers after being initialized the performance is fairly close. But as the list size grows, Linked List seems to take significantly longer than the Array List. At a list size of 30,000 it already starts to show the runtime difference between the two data structures. Once the list size reaches 100,000, adding 10,000 integers take more than a 3 second difference (3000 ms). Finally once the lists gets up to 190,000 items long, it takes the Linked List 14 seconds to complete an addition of 10,00 more items while the Array List stays below 1 second.

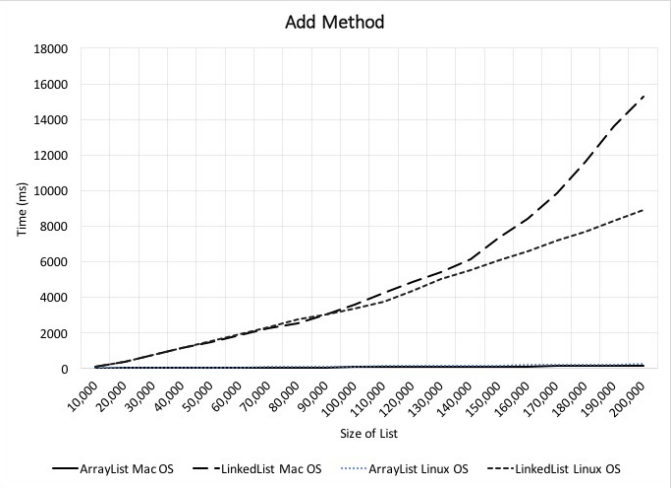

Fig. 1. Empirical results on the *add()* method of Java's Linked List and Array List using Mac OS and Linux OS

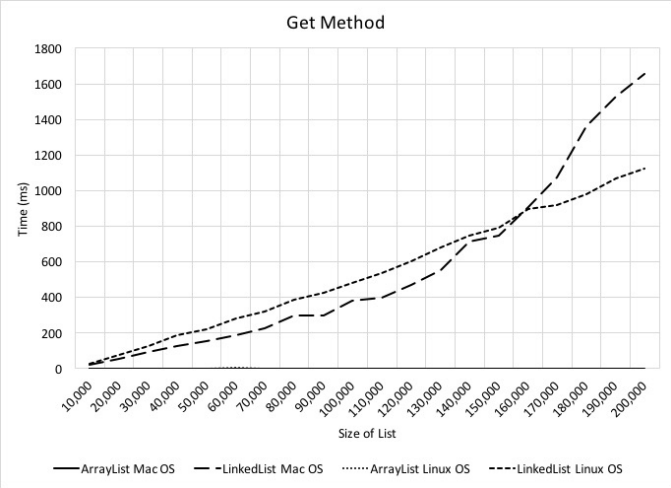

Fig. 2. Empirical results on the *get()* method of Java's Linked List and Array List using Mac OS and Linux OS

B. Linux OS

Besides the difference in processor and RAM between the two computers tested on, Linux OS and Mac OS handle system memory differently which directly affected the test results. Similarly to the tests on MAC OS when both lists initially add 10,000 integers after being initialized, the performance is fairly close. As the List size grows, Linked list seems to take significantly longer than the Array List. At a list size of 30,000 it already starts to show the runtime difference between the two data structures, same as Mac OS. Yet unlike Mac OS, while the Linked List grows the Linux OS seems to perform much better once the list size gets above 100,000. Finally once the lists gets up to 190,000 items long, it takes the Linked List only around 8 seconds to complete compared to the Mac OS time which was 14 seconds. Still, Array List is under 1 second throughout the entirety of the tests.

IV. Get Method Results

A. Mac OS

As shown in Figure 2 the empirical results for the *get()* method. Looking at the performance of Linked List vs Array List on Mac OS the data shows that after the lists are initialized to 10,000 calling the get() method 1,000 times only took the Array List 1 ms or 0ms, meaning the system couldn't register the timing in milliseconds. The linked list already started to take up to 50 ms to retrieve those items. After the list sizes reach 100,000 the Linked List starts to take 400 ms to complete the memory accesses while the Array List stays below 1 ms. Once the list size grows to 190,000 it takes Linked List 1.6 seconds to retrieve 1,000 random items, while Array List stays at below 1 ms.

B. Linux OS

Figure 2 also shows the performance of Linked List vs Array List on a Linux OS. Looking at the graphed data, the empirical analysis differs immensely from the asymptotic analysis of these data structures. Key points can be drawn about Figure 2 such as, after the lists were initialized at the start the Array List was still significantly faster, this was at size 10,000. As the list sizes start to grow the Linked List starts to take upwards of half a second above size 100,000. As said before the Array List is still taking under a millisecond to run 1,000 *get()* function calls because the memory cells are in order and can be accessed without looping through to the index that needs to be retrieved. The main difference between the Linux OS and the Mac OS test results lie in how the Linked List performed since the Array List performs quick enough to the point that it cannot be seen when graphed together. The Linked List performance on Mac OS grew consistently as the size of the list grew. On the Linux OS as the list got larger the performance scattered a bit and this could be due to other users taking up processing power.

V. Remove Method Results

A. Mac OS

In Figure 3 the empirical data of the *remove()* method for both data structures as well as both operating systems are graphed together. As said in Section II the *remove()* method will be tested by first initializing each list size to 200,000. Once this was complete the testing of *remove()* will start, removing 10,000 items at a time. As you can see in Figure 3 the Linked List immediately takes 12 seconds (12,000 ms) to complete the removal while Array List starts below 1 second (1000 ms). As the list size recedes and gets smaller the Linked List *remove()* method runs better yet still not comparable to the efficiency of Array List. Even when the list size gets to under 100,000 Linked List still takes significantly longer than Array List.

B. Linux OS

Looking at the Linux OS empirical results in Figure 3 it's shown that on the Linux OS, Linked List performed worse

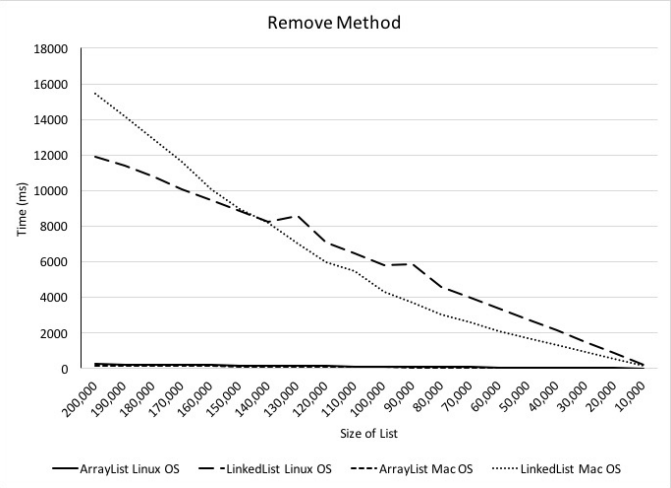

Fig. 3. Empirical results on the *remove()* method of Java's Linked List and Array List using Mac OS and Linux OS

	ArrayList	LinkedList
get()	O(1)	O(n)
add()	O(1)	O(1) amortized
remove()	O(n)	O(n)

Fig. 4. Asymptotic complexity of Array List and Linked List: *get()*, *add()*, and *remove()*.

than on Mac OS by 4000 ms at the start, which in the case of this particular test is size 200,000. As the list size gets smaller both the OS start to perform much more equally. Still on Linux OS Array List out performs Linked List by upwards of 15 seconds. When comparing this to how Array List performed on Mac OS there wasn't much of a difference, the difference was always within the range of plus or minus 50 ms. The main differences between these results has to do with what processes are running on the machine at the same instance, and how much RAM and cache the OS has to operate with.

VI. ANALYZING THE RESULTS

Figure 4 shows the asymptotic complexity of Array List and Linked List which will be compared to the empirical data gathered.

A. Add Method

First when looking at Figure 4, Array List seems to be overall a more efficient in terms of how each each method abstractly works. Within the *add()* method of Array List the asymptotic complexity as shown in Figure 4 does not factor in moving over $n - (index - 1)$ items. In terms of analyzing this method empirically this copying of data may slow down Array List as the list gets bigger. Yet, when looking at the results even factoring the copying of the data, it still out performed Linked List on both operating systems. Linked List has to go loop through to the index which on average takes much longer than Array List and this sheds light on some low level machine operations that make the difference. Moving large amounts of memory can actually be done incredibly fast to the point where it will be quicker than the amount of hops needed to loop through a Linked List to the specific index being added to.

B. Get Method

Figure 4 also shows the asymptotic complexity of the *get()* method for each data structure. Since the Array List can access a spot within the list in *O(n)* time it will consistently out perform Linked List because each item in the list are directly next to each other in memory. This allows the machine to skip over n items because each block of memory has to be the same size in an array. When working with data sets that contain upwards of 200,000 Linked List performs incredibly slow compared to Array List, looking at Figure 2 we can see that at size 200,000 Linked List took an upwards of 16 seconds to complete 1,000 *get()* method calls, while Array List consistently took either 1 ms or under a millisecond.

C. Remove Method

The asymptotic complexity of Array List and Linked List's *remove()* method are both *O(n)*, yet the Array List performed the removes at a substantially quicker rate. The reason lies within how memory is set up in computers which allows the copying of large amounts of memory and it can do this incredibly fast. As can be seen in Figure 3, the Linked List performed 4 seconds faster at size 200,000 than the Array List on the Linux OS. But as the list size becomes smaller the Linked List on Mac OS and Linux seem to share a similar trend. The differences can be attributed to how each particular operating system performs memory clean-up as well as memory allocation.

VII. CONCLUSIONS

After analyzing the asymptotic and empirical complexities of both Array List and Linked List in Java, a conclusion can be drawn that the implementation of an Array List will out perform an implemented Linked List. This is due to not only the restraints upon Linked List's data structure performance but also how memory is stored within an Array List allows for the machine to do the operations efficiently. Between the different operating systems there weren't huge differences in how the performance went. At a few points the Linked List did take longer which could be due to processes that other users are using on the server that the tests were being run on. On the Mac OS tests all other processes were slept so that Java could take 100% CPU power. Therefore, when efficiency is needed in software, and Array List will consistently outperform Linked List in at least three of the most important methods.

ACKNOWLEDGMENT

Special thanks to John Barr of the Ithaca College Computer Science Department for helping layout the testing as well as helping with the organization of the paper.

REFERENCES

[1] The Structure, Format, Content, and Style of a Journal-Style Scientific Paper. How to Write Guide: Sections of the Paper, abacus.bates.edu/ ganderso/biology/resources/writing/HTWsections.html.

[2] Instructions. Big O Cheat Sheets, cooervo.github.io/Algorithms-DataStructures-BigONotation/.

[3] ArrayList vs LinkedList vs Vector. ArrayList vs. LinkedList vs. Vector, www.programcreek.com/2013/03/arraylist-vs-linkedlist-vs-vector/.

[4] ArrayList. ArrayList (Java Platform SE 8), Oracle, 12 Jan. 2016, docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html.

[5] LinkedList. LinkedList (Java Platform SE 7), Oracle, 9 Oct. 2017, docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html.