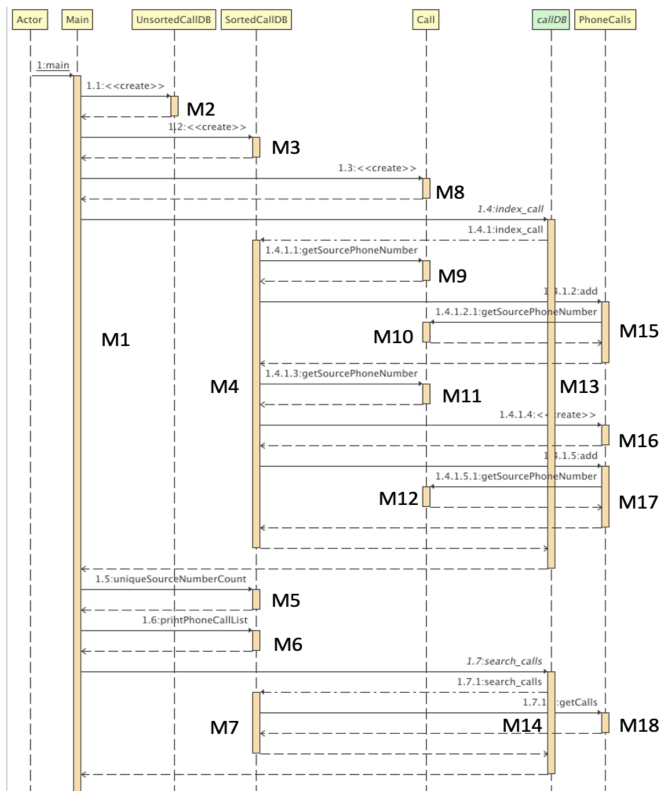
Mini Project #1 Report

Austen Skopov-Normane and HyeSeung Lee

Introduction

The overarching objective of this assignment is to experimentally test the efficiency of different algorithms. In doing so, we are also assessing our knowledge and understanding of different data structures that we have covered in class. At a smaller scale, this project ensures that we have a solid understanding of java syntax, java class structure, and efficient logic. We are then tasked with ensuring we understand how data structures work and how to implement them (i.e., lists). We then must apply this foundational knowledge to a “real-world” application, and further our understanding of how these data structures work at a higher level through empirical testing of compilation time. The data collection portion of the assignment is meant for us to really grasp the tangibility of the experiments, while the written portion is ensuring we truly understand the assignment from both a technical and broader standpoint.

Code Architecture



Method Key

Class: Main
- M1: Main method

Class: UnsortedCallDB
- M2: Constructor

Class: SortedCallDB
- M3: Constructor
- M4: index_call
- M5: uniqueSourceNumberCount
- M6: printPhoneCallList
- M7: search_calls

Class: Call
- M8: Constructor
- M9: getSourcePhoneNumber
- M10: getSourcePhoneNumber
- M11: getSourcePhoneNumber
- M12: getSourcePhoneNumber

Interface Class: CallDB
- M13: index_call
- M14: search_calls

Class: PhoneCalls
- M15: add
- M16: Constructor
- M17: add
- M18: getCalls

Figure 1: Sequence Diagram

Our coding solution works through the implementation of four classes and one interface. The Main class has the main method that is the entry point of our program for the user. The user inputs whether they want the requested information in an ArrayList or LinkedList and whether to receive this information in a sorted or unsorted manner (i.e., through either the SortedCallDB class or UnsortedCallDB class). The Main class reads from a file to load the calls into the selected data structure, and then it provides options for the user to search for calls from a selected phone number. Next, the Call class represents individual phone calls with given attributes (source phone number, target phone number, date of call, time the call started, and duration of call). This class also has getter methods that access these attributes to then be formatted into a string representation with a toString method. Further, the callDB Interface essentially serves as a blueprint for the SortedCallDB and UnsortedCallDB classes; it details the required functions of these classes: methods for indexing a call (index_call) and location calls from a specific phone number (search_calls). Moreover, the SortedCallDBClass implements the callDB Interface to manage the phone calls in a sorted manner. It has an inner class, PhoneCalls, to associate a source phone number with its phone calls. These PhoneCalls objects are managed using either an ArrayList or LinkedList. SortedCallDB provides methods for indexing calls, counting unique source numbers, printing the list of calls, and searching for calls. Lastly, the UnsortedCallDB Class also implements the callDB Interface, but it manages the calls in an unsorted fashion using a LinkedList or an ArrayList. This class also provides methods for searching for calls by a source phone number and indexing phone calls.

As seen above in Figure 1, the sequence diagram illustrates how the objects in the code solution interact through sequences of exchanged messages. The Actor is the user that initiates the main method, M1, that starts the flow of the program. Depending on user input, instances of UnsortedCallDB or SortedCallDB are created, as depicted by M2 and M3. M8 represents an instance of Call being created with details from the file the user provides. The index_call method of either the SortedCallDB or UnsortedCallDB Class is called to add the Call instance to the database through M4, M9, M10, M11, M12, M13, and M14. Furthermore, M15 and M17 represent if the SortedCallDB Class is used, where the add method can be implemented to add the Call object to the appropriate PhoneCalls object. Then, the search_calls method is used to search for all calls generated from a particular source phone number, as represented by M7 and M14. Additionally, M16 represents the PhoneCalls class constructor, which is called as a part of managing the calls in SortedCallDB. Finally, the getCalls method retrieves all calls associated with a particular source number in SortedCallDB, as illustrated by M18.

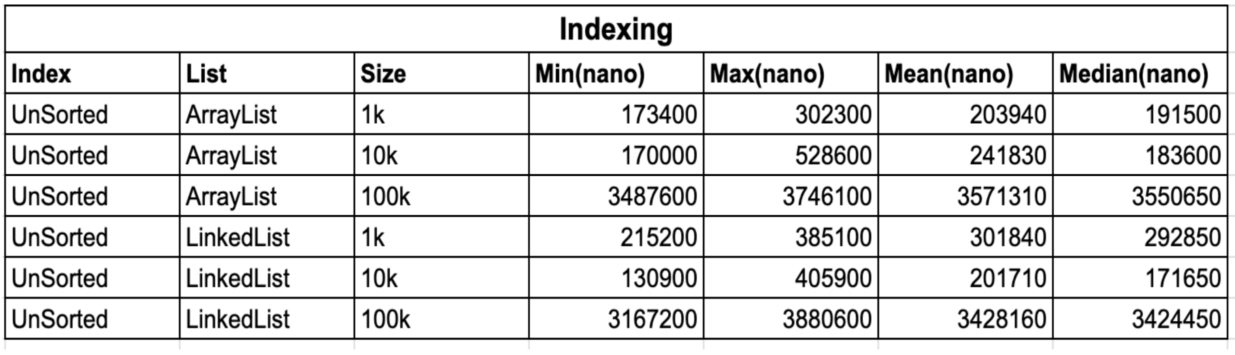
Computer Specs

For this experiment, the data collection was done on one laptop. Here are some of the device specifications. The processor, which is an Intel(R) Core(™) i7-8565U CPU@ 1.80GHz 1.99 GHz. The installed RAM of 16.0GB(15.7 usable). The system type is a 64-bit operating system with a x64-based processor. The edition is a Windows 11 Home, and the manufacturer is

HP. The SSD is KXG60ZNV256G by Toshiba, and the GPU is Intel(R) UHD Graphics 620 with the total available graphics memory of 8171MB. Lastly, the Java version used is version 17.0.9.

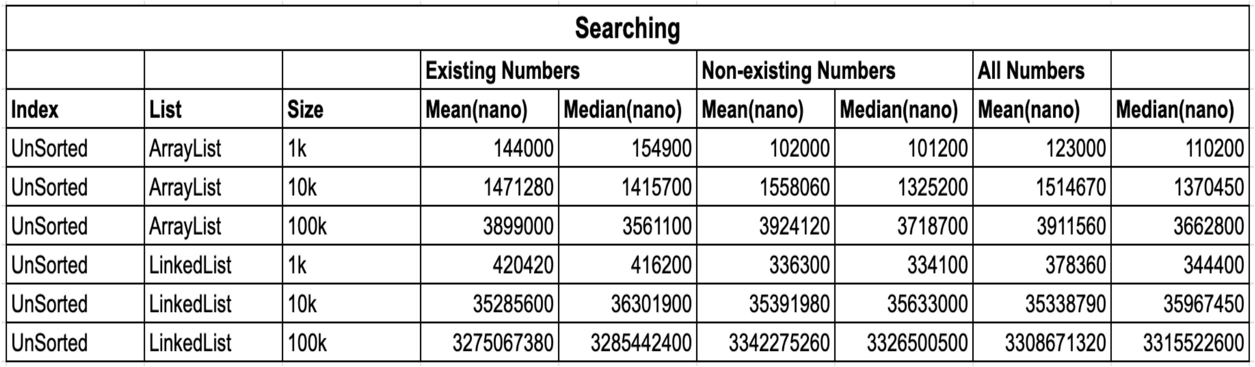
Results Part 1. UnsortedCallDB

Our version of indexing and searching for UnsortedCallDB follows the instructions of the project description; the call is inserted at the end of its internal unsorted list of calls for index_call, and we also implemented sequential search for search_call. The main purpose of this class is to perform these two methods when the system uses the unsorted data type. The parameter types and return types make sure we use uniformed language throughout the program in order to keep the structure of this program organized and clear. Please note that all run time results are in nanoseconds.



Indexing						
Index	**List**	**Size**	**Min(nano)**	**Max(nano)**	**Mean(nano)**	**Median(nano)**
UnSorted	ArrayList	1k	173400	302300	203940	191500
UnSorted	ArrayList	10k	170000	528600	241830	183600
UnSorted	ArrayList	100k	3487600	3746100	3571310	3550650
UnSorted	LinkedList	1k	215200	385100	301840	292850
UnSorted	LinkedList	10k	130900	405900	201710	171650
UnSorted	LinkedList	100k	3167200	3880600	3428160	3424450

Figure 2: Results table with the requested metrics for indexing



Searching								
			Existing Numbers		**Non-existing Numbers**		**All Numbers**	
Index	**List**	**Size**	**Mean(nano)**	**Median(nano)**	**Mean(nano)**	**Median(nano)**	**Mean(nano)**	**Median(nano)**
UnSorted	ArrayList	1k	144000	154900	102000	101200	123000	110200
UnSorted	ArrayList	10k	1471280	1415700	1558060	1325200	1514670	1370450
UnSorted	ArrayList	100k	3899000	3561100	3924120	3718700	3911560	3662800
UnSorted	LinkedList	1k	420420	416200	336300	334100	378360	344400
UnSorted	LinkedList	10k	35285600	36301900	35391980	35633000	35338790	35967450
UnSorted	LinkedList	100k	3275067380	3285442400	3342275260	3326500500	3308671320	3315522600

Figure 3: Results table with the requested metrics for searching

The two different types of lists used in this step are ArrayList and LinkedList. Let's consider indexing first. On figures 2 & 3, we can see the comparison by looking at the median of each 10 tests that were done on the same variation of types. However, it is hard to spot the relationship between the two variations. For example, if we compare Unsorted-ArrayList-1k and Unsorted-LinkedList-1k (figure 2), we can see that the ArrayList type is faster by 101350 nanoseconds, which is not a significant difference considering the many variable factors. When we consider the rest of the results (10k and 100k), we see that ArrayList actually takes slightly longer than LinkedList, but again, this difference is not significant in actuality.

Considering one of the characteristics of an ArrayList, we know that insertion can be slow because it requires shifting elements. Thus, everytime we insert an element in a list (e.g. at the first index), then we need to shift everything that is behind that index by one. With the resulting increase in input size, the amount of time it takes to shift every input will increase as well. In this particular case, we only operate insertion at the end, due to the nature of this project. This function of ArrayList has the time complexity of O(1), because adding an item at the end of the list does not require shifting.

On the other hand, the insertion of LinkedList can be fast at the beginning or end of the list with the access to the first and last node with pointers, but it becomes progressively slower as it gets closer to the middle of the list. This is because we need to traverse through the list until we reach the desired position.

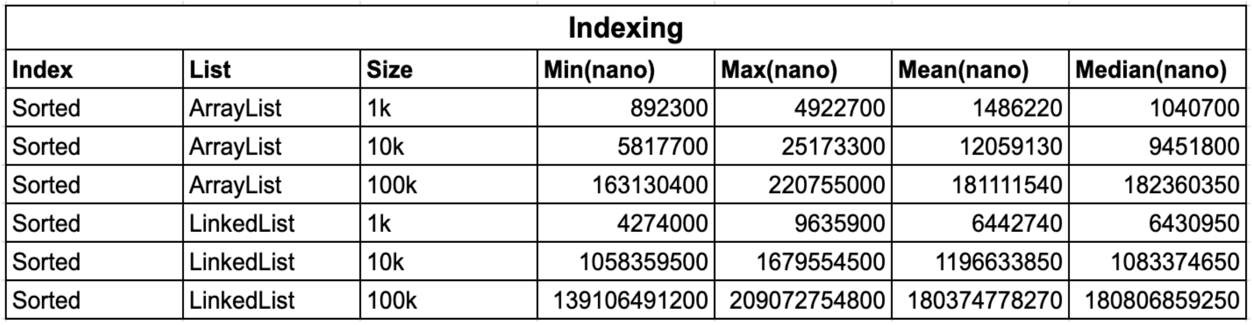
With searching the calls (figure 3), we see a clear relationship amongst the numbers when comparing them with the fixed data structure type and input size. LinkedList's search is not only slower in comparison to ArrayList, but as it increases in input size, the search takes progressively longer. This result was expected, because ArrayList provides index access; this is considered constant time, O(1), whereas LinkedList needs to do traversing to get to the desired position. As long as the desired item is sitting at a random position in the LinkedList, it will take longer for the LinkedList to do the traversing.

The size of the input file that we used in these tests are 1k, 10k, and 100k. Let's again consider indexing first. Comparing the run time of UnSorted ArrayList with file sizes varying greatly, we see some unexpected correlation amongst the numbers (figure 2). With the increase in file size from 10k to 100k, we can see that the median time the program took to finish each procedure had significantly increased. However, when we looked at the data from 1k to 10k, we discovered it decreased. We think that this case is specific to the ArrayList type. When the array is full and needs to be resized, a copy operation is performed. The time complexity for the specific insertion becomes O(n), where n is the number of elements in the ArrayList. We are predicting this copying operation occurred in the test and caused the increase in time spent for the tests. As we took a closer look at the LinkedList's case, we discovered a pattern: both case's 1k vs 10k have a small amount of decrease. We took another look at other data that might be relevant, especially the minimum and maximum values of UnSorted ArrayList 1k vs 10k, and UnSorted LinkedList 1k vs 10k. The minimum values are outliers in both cases; the minimum values for both 10k tests were smaller than the minimum for 1k tests. The tests were severely skewed due to these outliers, and the cause of these outliers could be the result of the JVM warm-up, CPU caching, or garbage collector activities.

For searching, we discovered that the increase in input size increases the amount of time spent on doing the search operation, and this applies to every variation of the UnSorted tests. Even when comparing existing and non-existing numbers individually, they are all proportionally affected by the increase in input sizes. So input size does matter in this program.

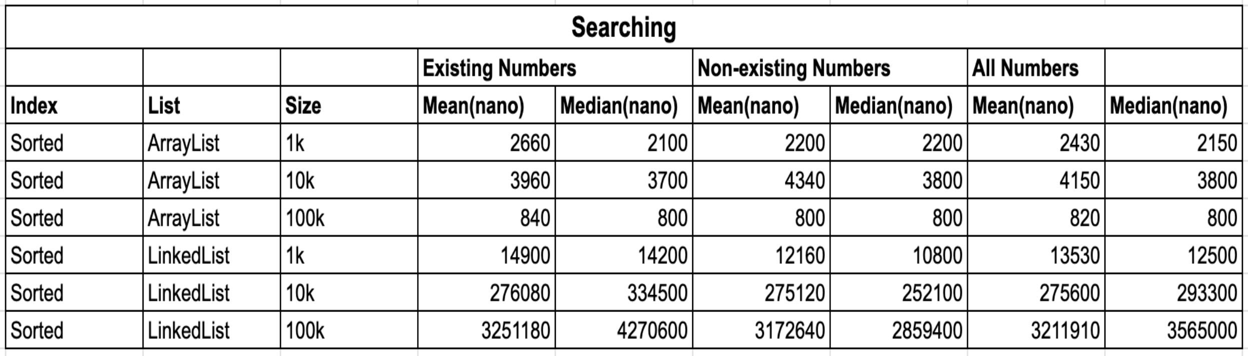
Results Part 2. SortedCallDB

Our version of indexing and search for SortedCallDB also follows the instructions closely. Our constructor is used to keep a valid structure of this function. The index_call function uses binary search structure to ensure the efficiency of the function. It checks if the PhoneCalls object exists for x.source_number first using binary search. Then, x is added to that object if the object is found, and sorted insertion is implemented in case a new PhoneCalls object is created. Afterwards, binary search is used again to search for all relevant calls.



Indexing						
Index	**List**	**Size**	**Min(nano)**	**Max(nano)**	**Mean(nano)**	**Median(nano)**
Sorted	ArrayList	1k	892300	4922700	1486220	1040700
Sorted	ArrayList	10k	5817700	25173300	12059130	9451800
Sorted	ArrayList	100k	163130400	220755000	181111540	182360350
Sorted	LinkedList	1k	4274000	9635900	6442740	6430950
Sorted	LinkedList	10k	1058359500	1679554500	1196633850	1083374650
Sorted	LinkedList	100k	139106491200	209072754800	180374778270	180806859250

Figure 4: Results table with the requested metrics for indexing



Searching								
			Existing Numbers		**Non-existing Numbers**		**All Numbers**	
Index	**List**	**Size**	**Mean(nano)**	**Median(nano)**	**Mean(nano)**	**Median(nano)**	**Mean(nano)**	**Median(nano)**
Sorted	ArrayList	1k	2660	2100	2200	2200	2430	2150
Sorted	ArrayList	10k	3960	3700	4340	3800	4150	3800
Sorted	ArrayList	100k	840	800	800	800	820	800
Sorted	LinkedList	1k	14900	14200	12160	10800	13530	12500
Sorted	LinkedList	10k	276080	334500	275120	252100	275600	293300
Sorted	LinkedList	100k	3251180	4270600	3172640	2859400	3211910	3565000

Figure 5: Results table with the requested metrics for searching

The list types ArrayList and LinkedList definitely have a significant impact on the Sorted indexing and searching. On figures 4 & 5, we discovered that there is a relatively consistent increase in run time when it is done with LinkedList. LinkedList's runtime was not only slower at first, but it was consistently becoming slower as the data grew.

For indexing, since we used binary search for indexing for both LinkedList and ArrayList, let's compare the structure of each list. As we mentioned before, ArrayList has constant-time access to a random index, whereas for LinkedList, we need to traverse through the list to get to a certain position. In other words, ArrayList has a O(1) notation for accessing, and LinkedList has O(log n) notation for accessing.

In regards to searching, ArrayList has a O(log n) notation for binary search since we decrease the input size by one more than half. On the other hand, LinkedList has to traverse through the list to get to a certain index; therefore, it has a O(n log n) notation for binary search.

We expected to see a consistent increase in the resulted run times with the increase in input sizes, but there was an exception. Most of the data gathered for the Sorted data structure shows that the lager the input size, the longer it takes to complete for both indexing and searching. However, when we took a look at Sorted ArrayList 100k compared to the other 2 sizes, we discovered that there is a significant decrease. Moreover, the mean and median for existing and non-existing numbers are very similar, especially considering that the test run times are in nanoseconds. Thus, there are a few possibilities for the cause of this behavior. Most of them are closely related to computer systems and Java programming language, such as garbage collector, CPU cache, etc. After consultation, we decided that Java optimization was the most reasonable cause for this behavior, since the means and the medians for existing numbers, non-existing numbers, and all numbers were very similar (as shown in figure 5).

General Analysis

Comparing Sorted and UnSorted lists, we can see that the results vary significantly, which means the data structure type plays a significant factor in the results of this specific program. For indexing the calls, when we compare the results from figure 2 and 4, we can see that Sorted ArrayList tests take longer than UnSorted, and Sorted LinkedList tests take longer than UnSorted; this is because of the general structure of these functions. For UnSorted Indexing, we are doing the ".add()" operation, which is considered as amortized constant time, O(1), which means that most of the time, adding an element to the end of an "ArrayList" takes a constant amount of time regardless of the size of the list. However, as we mentioned earlier, ArrayList needs resizing. This involves copying all the elements from the old array to the new one, which is a linear time operation O(n). Considering that the resizing doesn't happen too often, the average time complexity of indexing should still be close to constant.

As for LinkedList, we know that we have access to the first and the last node due to LinkedList nature. Therefore, when we are adding an element at the end of the list, there is no need to traverse through the list, so it has a notation of O(1) as well.

In regards to searching, when we compare the results from figure 3 and 5, we find that Unsorted arraylist takes longer than Sorted, Unsorted LinkedList takes longer than Sorted. This is partially due to the structure of these functions. We used a sequential search for the UnSorted search, and a binary search for Sorted search. Binary searches will be faster in most cases because the input sizes are getting halved (almost) after each loop is run through, whereas for a sequential search, the run time will be directly impacted by input size. Because it is plainly going over every element and only stopping once it finds the desired element, with the more elements to go through (input size), the longer it takes.

Types of lists also matter in this particular program. As we have explained before, ArrayLists and Linkedlists have very different structures. To summarize briefly, ArrayLists have constant-time access to random indexes, thus having fast access to random elements. However, when it comes to insertions and deletions, because of the shifting nature of ArrayLists, the process can become slow when it's done at the beginning and/or at the end. We also have to consider the resizing nature of ArrayList. On the other hand, LinkedLists have access to the first and the last element of the list, therefore insertions and/or deletions at the beginning and/or the end are fast. Since it requires traversing for random access, insertions and/or deletions at random indices and accessing random elements can be slow.

As a part of our discovery, the impact of computer specs in particular was very interesting to us. The Windows(C:) hardware on our laptop is the KXG60ZNV256G by Toshiba, which is an SSD (solid state drive). These drives use NVMe(non-volatile memory express) technology over the PCIe interface, which allows for much faster data transfer HHDs.

The GPU on this laptop is Intel(R) UHD Graphics 620 with the total available graphics memory of 8171MB, while the shared system memory is 8043MB. While the GPU does not directly affect the compilation speed of JAVA programs, its impact is more about ensuring the system remains responsive and usable during these tasks, rather than speeding up the compilation process itself. Since the GPU on this laptop uses shared system memory, the available RAM for both the GPU and the Java applications is from the same pool. With 8171 MB available to the GPU, this is beneficial for running memory-intensive Java applications or multiple applications simultaneously in IntelliJ.

Additionally, background program count while conducting data collection was 129, while the CPU usage (including IntelliJ and background programs) was around 30%-60%. The memory usage of these programs was about 75%. We monitored the CPU and memory usage during the majority of the test runs to ensure this would not impact the results significantly.

Difficulties

We faced a few difficulties while implementing this project. Firstly, with more freedom in being able to add auxiliary functions, we had some trouble deciphering which auxiliary functions would be helpful in our project; we eventually landed on using an interface class to assist with indexing and searching calls. This proved helpful in avoiding code repetition, but was tricky to figure out how to do so properly. Secondly, in testing our search functionality, we ran into some issues with the "next line" function of our scanner, as we had to spend some time debugging to figure out the issue, which was that the method was improperly reading the next line as an empty line instead of our code. Lastly, we had trouble with indexing the sorted calls. We were unnecessarily adding in steps that lead to inefficiency and lengthy indexing time. Through some guidance, we learned why our original logic was inefficient, and how to implement a more efficient way to get the data we want. Overall, we faced some minor challenges that we were able to overcome with careful examination and double checking that our logic was truly efficient.

New Application

A new application where one would use lists would be implementing a health and fitness tracker to help people manage and improve their overall health, wellness, and fitness. This application would consist of both tracking features and fitness goals for the user to have access to and is composed of four components. One piece (class) of the application would be a workout tracker. This would provide users with a list of exercises that have different features such as workout type (strength training, cardio, etc.), duration of workout, and average calories burned during a certain type of exercise given its duration. Another class would be a diet tracker that lists daily food and drink consumption within four categories: breakfast, lunch, dinner, and snacks. Each food and beverage would have nutritional information like the item's calories and content (vitamins, carbohydrates, proteins, etc.). The third class would track a user's health metrics, like providing a list of a user's weight, height, and blood pressure. Users could then see the changes in their metrics over time. Lastly, the fourth class would consist of a list of health and fitness goals. Some of these goals could be exercising a certain amount of times per week, losing weight, or lowering blood pressure. The application would provide motivational printing statements based on the user's goals.

This application would be set up by the programmer, and the users would input and interact with the application. As the programmer, we would create foundational structures that would enable the user to input their health and fitness data. From the programmer's side, there would be class structures for different types of data such as workouts, meals, health metrics, and goals -- as aforementioned. Each class would have their own specific attributes (e.g. a "Workout" class would include attributes for workout type, duration, and calories burned). The programmer would also manage lists through various methods, such as adding, removing, and updating items in the lists. The lists would be organized through built-in sorted functions based on criteria such as goal progress and dates. This new application would also prompt numerous input fields for users to enter new data into different logs, edit and delete existing data, and see their current data in a human-readable format. Lastly, the program would also consist of statistical summaries and comparisons of their metrics against the goals they set. The statistical summaries would include things such as their weight gain/loss, average workout duration, average calories consumed per day, etc.

In terms of what the user would input, they would be able to enter the details of their daily workouts, given the attributes of the workout class (type of workout, duration, and calories burned). The user would also enter their meals for the day with metrics like what food they ate for each meal and the meal's nutritional information. Additionally, the user would be able to set goals for their target outcomes, such as specific weight loss or gain amounts, target blood pressure, workouts per week, etc. The user would then be able to review all of their logged data, as well as edit or delete anything.

Lists are vital for this proposed application. First of all, in order to store this large quantity of flexible data, an organized "container" that can be changed (is dynamic) is necessary. The health and fitness tracker needs to be "resized and changed" for specific users. Other data structures would be inferior and less efficient for this application, specifically arrays. Arrays have a fixed size, which would not be useful in this case, where users would be changing the inputted data and therefore the size of the data. Inserting and removing items is also quicker to do with lists compared to arrays. Health and fitness levels/goals vary per person, and lists allow for easy insertion, removal, and modification of entries (new fitness goals, new workouts, etc.). The flexibility and personalization of our application make lists crucial. Lists also accommodate for changing scalability, as over time in this application, the user would be inputting more and more data. Additionally, lists are a very organized data structure that have a sequential order. This health and fitness application is monitored over time; user's can see their data in a chronological manner so that trends can be made and goals can be worked up to throughout time. Additionally, lists allow for quick access to data points. Since our program would also include statistical summaries, lists could easily access the data to compute these summaries. Lists are inherently great for personalized trackers and logs, due to its aforementioned functions, making them a necessary data structure for this application.

Sorted lists will be incredibly useful in the health and fitness tracker application. If the lists are sorted, then users can quickly find the relevant data they are looking for. Specifically, their workout logs, health metrics (over time), and progress towards their goals would be sorted. This way, a user can pull their data from a specific day or a specific workout, allowing for efficient analysis of data. If the lists were not sorted, then the user would not be able to easily access the desired data, rendering this application inefficient.

Conclusions

Overall, this project allowed us to see the efficiency of different algorithms using timed data. We were able to note how our code worked holistically using a sequence diagram. More specifically, using different list data structures, we were able to see stark differences between the efficiencies of parts of our code through UnSorted searching. We noticed differences when comparing sorted and unsorted lists as well, such as Sorted searching. Through final analysis of our data, we concluded that the use of ArrayList is more efficient than LinkedList with some minor exceptions. This project allowed us to think of a new application where we could use lists, a health and fitness tracker, which would be an exciting project to pursue to further analyze the efficiency of different algorithms and code logic.We did face a few minor challenges, such as having difficulties choosing a more efficient structure for the purpose of speeding up data collection process. However, this project allowed us to learn how to thoroughly analyze the logic and efficiency of different types of data structures and how to correctly implement lists.

Furthermore, we learned about how other factors like computer specifications can impact our data and the importance of running multiple tests to get the most accurate data as possible. It is also crucial to pay attention to the details such as the outliers in the data and what they tell us about the program. All in all, this project gave us a more in-depth understanding of data structures in java.