RUN-TIME EFFICIENCY ASSESSMENT FOR THE SELECTION OF OPTIMIZED DATA STRUCTURES FOR JAVA PROGRAMS

A PROJECT REPORT SUBMITTED BY

R.N.G.J.H. NAWARATHNA
(S/16/417)

to the

DEPARTMENT OF STATISTICS AND COMPUTER SCIENCE

in partial fulfillment of the requirement

for the award of the degree of

BSc. (Honors) in Computer Science

of the

UNIVERSITY OF PERADENIYA
SRI LANKA
2022

DECLARATION

I do hereby declare that the work reported in this project report was exclusively carried out by me under the supervision of Mr. Prabhath Gunathilake. It describes the results of my own independent work except where due reference has been made in the text. No part of this project report has been submitted earlier or concurrently for the same or any other degree.

Date: ……. …….

Signature of the Candidate

Certified by:

1. Supervisor: Mr. Prabhath Gunathilake

Date: ……. Signature: …….

2. Head of the Department: Dr. Sachith Abeysundara

Date: ……. Signature: …….

RUN-TIME EFFICIENCY ASSESSMENT FOR THE SELECTION OF OPTIMIZED DATA STRUCTURES FOR JAVA PROGRAMS

R.N.G.J.H. Nawarathna

S16417@sci.pdn.ac.lk

Department of Statistics and Computer Science, University of Peradeniya, Peradeniya, Sri Lanka

Data structures help programmers a lot in terms of organizing, storing, and handling data more efficiently. The set of built-in optimized data structures has many advantages when it comes to Java because they can be used depending on the programmer's needs. These include data structures like Arrays, Lists, Maps, etc. This particular research work intends to make those programmer's needs accessible more efficiently by providing guidance or assistance by comparing and analysing the run-time behavior of different Java data structures with a machine learning approach. We have taken the List, and the Map interface from the Java Collection interface into account to carry out the research. By having the selected data structures benchmarked to get the run-time behaviour in the sense of how a data structure in a particular program manages to handle the memory and the run-time, we are able to get an idea of how that data structure would perform in different scenarios. Java data structures have properties like growing policies; the data structure would grow as it gets filled with elements and dynamically increases and decreases the storage to handle the input data efficiently. With the collected data of the run-time behaviour of those selected data structures, we can extract the information of so-called data-structure properties. We could differentiate those data structures in terms of their run-time behaviour by analyzing the data. Furthermore, have an assisting report to suggest better data structures to use particularly in programs like Java enterprise applications, to improve the performance and ultimately give a better experience to the end-user.

ACKNOWLEDGMENTS

First and foremost, I would love to express my deep and sincere gratitude to my supervisor, Mr. Prabhath Gunathilake, for giving me the idea and supporting me throughout the entire research work while providing his invaluable guidance and supervision amidst his tight schedule. I cannot imagine how hard it would be to complete the research if it weren't for his wonderful supervision. It was a great honour to work and study under his profound guidance, enclosed with great insight. I want to extend my heartfelt gratitude for his outstanding support. I believe I am greatly indebted to his extraordinary endeavour and support throughout my research work and in regard.

Also, I would like to thank the Department of Statistics and Computer Science staff for sparing their valuable time and giving suggestions and advice to carry on the research.

I am extremely thankful to my parents and siblings for their care, sacrifices, and prayers and for supporting me to complete this research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CPU: Central Processing Unit
CSV: Comma Separated Values
I/O: Input-Output
MB: Megabytes
ms: Milliseconds
UUID: Universally Unique Identifier
RAM: Random Access Memory
SSD: Solid State Drive
VM: Virtual Machine

CHAPTER 1
INTRODUCTION

Since the beginning of software development, data structures have played a significant role in almost every program. Those provide the right way to organize information in the digital space and are primarily used in machine learning, operating systems, graphics and many more. For developers to develop performance-based productive enterprise applications, data structures are a must to know. When it comes to software performance, there are major things to consider, such as the memory that particular software will use during its run-time. Hence, to manage all these carefully, data structures help programmers a lot; in fact, programmers cannot be called good programmers if they do not better know data structures. Even though there are many tools available to help improve programming skills and increase performance, data structures specified tools are less to look into, even if it should be a significant concern.

With the tech development in recent years, computer hardware and software have become more emerged within one another. As it continues to develop, the thing that matters is the performance of those applications with the relevant hardware. Performance is one primary concern when it comes to the quality of software. For example, in a business application, it directly affects the revenue of that business. The Quality of Service (QoS) provided by these enterprise applications can be identified as performance, availability, and security. The proposed system here is mainly focused on the performance of these application programs.

Over the years, Java has become one of the major languages for developing enterprise applications. Java has helped developers to create large scalable applications more securely. Developers choose Java becuase it is more secure, reliable, and powerful enough to create such complex enterprise applications. People often talk about the performance of these applications and how they can be improved in terms of performance. Performance models to predict performances and suggest necessary changes in the application are reliable ways of improving software quality. Also, this helps the software engineering industry bring higher efficiency and better quality to the software that they build.

1.1 Problem Statement

In software, data structures are a significant concern because they help organize and store data so that developers can perform operations on data more efficiently. These are used in almost every application in various types such as Arrays, Maps, Linked Lists, Stacks, etc. Data structures play a huge responsibility in terms of the performance of so-called applications. So, when it comes to developing software, developers use multiple data structures. There can be situations that they might think at a particular point that the data structure used is good; however, it is not afterwards. The only way to find out this is by measuring the performance of the software.

Developers use various software testing mechanisms to ensure their applications perform well in the consumer's hand. However, often, these testing mechanisms are not ideal in most cases. Building an exact system for real-world application testing is hard to achieve since it requires too much money and time. Sometimes it is not easy to test software like this because reproducing certain functionalities is challenging and time-consuming.

Performance modeling comes in handy to help developers predict their applications' performance and improve the quality. Performance models provide performance predictions rather than performance observations. These performance predictions are beneficial to identify performance bottlenecks and to timely anticipate future performance problems.

As a vastly grown platform, Java uses many advanced data structures a lot and helps developers create more complex enterprise applications for consumers. Often, performance may vary from one data structure to another in an application depending on the use case. Developers do not usually measure performance when the application runs from time to time because it is time-consuming. So, what the proposed model does is that it calculates the performance of a particular Java program when it gets executed and analyses those performances. After that, it suggests what data structure can be used instead to increase the performance immediately. It relies on the performance of the particular program and hence more accurate the prediction is. The model is more like a software quality assurance component that makes suggestions at the data structure level to improve the software performance.

1.2 Aim of the Project

Analysing the run-time behaviour of Java applications that use data structures in the implementation, and suggest if there are better, available data structures to use instead.

1.3 Objectives of the Project

1. Develop a program solution to store information of run-time behaviour of Java data structures from the Java List and Map interfaces in the Java Collections interface.
2. Identify the best possible alternative data structures for a Java program using machine learning within the chosen data structures.
3. Propose a system to Integrate the information of run-time behaviour and generate an assisting report.

CHAPTER 2
LITERATURE REVIEW

Software Performance Engineering (SPE) is a significant category under software engineering. As it is defined, Software Performance Engineering (SPE) represents the entire collection of software engineering activities and related analyses used throughout the software development cycle, which are directed to meeting performance requirements [1]. The performance of the software is one of the main factors deciding the product's success.

Resources are required to keep a system running within the parameters of the non-functional performance requirements. The performance of a system is decided by how the system interacts with the resources, and we can define resources as hardware (CPU, bus, I/O and storage, network), logical resources(buffers, locks, semaphores), processing resources (processes, threads) [1]. As the paper [1] describes it, a determining factor for performance is that resources have a limited capacity, so they can potentially halt/delay the execution of competing users by denying permission to proceed. There are several SPE activities; Identify concerns, Define and analyze requirements, Predict Performance, Performance testing, Maintenance and evolution, and Total system analysis [1].

Several types of research have been done related to performance modeling over the past years. According to Wu and Woodside [2], component-based software engineering brings efficiency and quality to software development using reusable and configurable software components. Hence, it helps to increase the quality of performance engineering. Also, they suggest that performance should be based on the pre-defined software component. So, the same approach can be used to build a performance model for Java applications as well. According to research named Performance Monitoring of Java Applications [3], the performance monitoring system should monitor the following elements of execution behaviour.

1. The invocation of methods
2. Object allocation and release.
3. Thread creation and destruction.

4. Mutual exclusion and cooperation between threads.

The results should include attributes that can be used to calculate performance measures [3]. These data might be stored in a database and accessed through an application programming interface (API). When it comes to monitoring, there are two types: event-driven and time-driven. Time-driven monitors at certain time intervals, whereas event-driven observes events in a system [3].

In terms of Java performance monitoring, there are a few tools that we can use. Visual VM [10], JVM [9], JMeter [8], Javasisst [11], and Sentry [7] are those tools that fall under mentioned tools. We can use these tools to measure the performances of various Java applications and hence get an initial idea about the performance of a particular program. Other than these, Java Profiler is a tool that monitors Java bytecode constructs and operations at the JVM level. This tool is also important because a running code is not enough for production applications. So, programmers need to look at memory allocations to see how the product works. For that, Java Profiler is a good tool. The mentioned tools Java Visual VM falls into this category.

Performance benchmarking is another interesting method of collecting data about any program, including Java programs. Benchmarking includes gathering and comparing quantitative data, and it comes in handy for understanding the performance gaps. This process can be done in several ways, such as benchmarking a list of programs to collect data or creating a dedicated program for the purpose of benchmarking smaller components of a particular program. When it comes to this particular case, those two methods can be used depending on the situation. There are two parts of the run-time behaviour of a program; memory consumption and actual run-time. Particularly in Java, there are libraries and built-in functions available to achieve these tasks. To measure the run-time or the memory consumption taken by a data-structure process in a particular Java program can be measured using built-in functions like *System.nanotime()*. The Java garbage collector can also be used to measure memory consumption.

Platforms like Github provides thousands of freely accessible open-source programs to test our theories. But, sometimes, those programs do not show much performance differences in the benchmarking results. Because it is nearly impossible to find any real-world, large-scale multi-

tiered enterprise applications that use things like data structures; that affect the performance of the whole application heavily. Implementing a dedicated program to test out such components is also a suitable method. Larger datasets can be used to handle data structures, and the performances can be measured depending on the need. For example, such benchmarking solution can be implemented using the data structures that we need to perform the benchmarking on and have a separate large random data set to try out various functions from data structures like *get(), put(), remove(),* etc. Afterwards, the run-time and the memory consumption can be measured as those data continuously gets processed through so-called functions [12].

Java is a program that uses data structures a lot to get the work done. Measuring the overall performance of an application includes data structures that have been used in that application as well. A data structure is that performance tasks efficiently organise, process, retrieve, and store data. Depending on the approach for the program, the data structure that needs to be used varies. Hence the application's performance varies as well.

In most cases, sorting algorithms are used inside the implantation of the data structure. The choice of sorting algorithm depends on various parameters like the amount of memory or machine time needed for running a program, how fast and accurately it sorts a list [6]. Even though developers have the tools mentioned above, there is still a lack of suggesting tools to measure the performances at the program run-time and recommend better data structures to use instead. So, considering all these things, a unique application can be proposed to measure the performances of Java programs that use data structures and suggest various data structures depending on the usage inside the application.

CHAPTER 3
METHODOLOGY

3.1 Idea Overview

Java Collection framework provides plenty of interfaces that hold the roots of well-optimized data structures. For this research work, two interfaces have been selected with altogether six data structures. Those interfaces are the List and the Map from the Java Collections interface. List interface contains LinkedList, ArrayList, and Vector, while the Map interface contains HashMap, TreeMap, and LinkedHashMap. All these selected optimized data structures have unique features, including their behaviour in the run-time.

Having benchmarked the mentioned data structures from both the interfaces as two groups, run-time and memory usage data can be collected separately for further analysis. Moreover, we can use collected data to get information about how each data structure would behave in terms of run-time and memory consumption. Afterwards, we can use that information to differentiate between data structures depending on the developer use case and achieve better performance in the programs overall.

3.2 Standard definitions related to data structures

3.2.1 *hashCode()*

A Java function that returns an integer or a 4 bytes value generated by a hashing algorithm. That unique hash code is assigned to an object or attribute, enabling quicker access.

3.2.2 *equals()*

Equals function is used to compare two objects. It compares the values associated with particular objects, and the function will return true if the objects are located in the same memory location.

3.2.3 The contract

If the *equals()* function returns true for two objects, then the hash code of both objects should be the same. But, it does not mean that having the same hash code for two objects should always return true according to the *equals()* function.

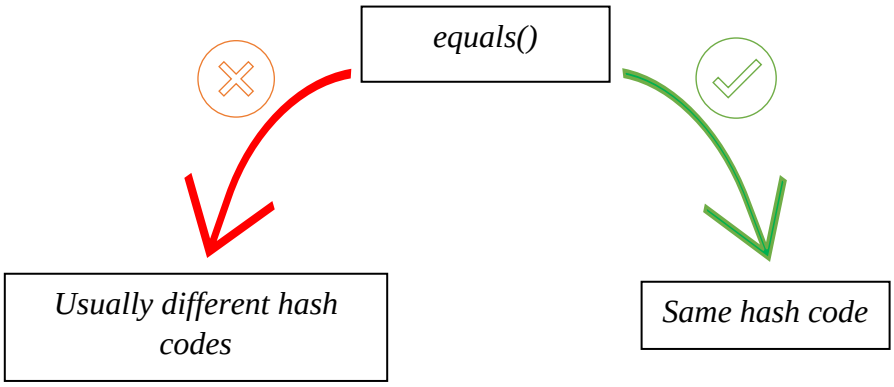

Figure 3.1: The contract between *hashCode()* and *equals()* functions.

3.2.4 Initial Capacity

Initial capacity helps with the memory allocation for a particular data structure. When we define an initial capacity, the memory spaces gets allocated according to the define number. Otherwise, it will use the pre-defined default value as the initial capacity. When a data structure gets filled and reached this capacity, the rehash operation takes place to increase the current capacity by an amout of current capacity is divided by the load factor [3.2.6]. However, setting the initial capacity should be done wisely because it could affect the performance.

3.2.5 Rehashing

Data structures increase their capacity dynamically as more elements are put into them. Especially in map data structures, when the capacity increases after reaching the threshold value, the hash codes for all the existing elements are recalculated and assigned to match the bigger capacity.

3.2.6 Load Factor

The load factor is the measure that decides when to increase the capacity of a data structure.

$$Load\ Factor = \frac{Total\ numbers\ of\ items\ stored\ before\ the\ increment\ happens}{Size\ of\ the\ data\ structure}$$

3.3 Selected Data Structures

Each data structure has its own unique features regarding growing policies and how it behaves in the run-time. Six common functions have been taken into account for the benchmarking process, and those are *insert(), insertAt(), contains(), indexOf(), get(),* and *remove(). insert()* is the function(usually called *add(E e)* in the List interface, and *put(K keyValue, V value)* in Map interface according to Java doc.) that helps input the data into the data structure. In contrast, *insertAt()*(usually called *add(int index, E element)* or *replace(K key, V value)* in Java doc.) helps to replace a particular element. *contains()* and *indexOf()* functions are basically search functions while *get()* being the functions that help to retrieve elements from the data structure. As it appears by the name, *remove()* helps to delete items from the data structure. All those functions are common and mostly used functions between all the selected data structures. Furthermore, the selection of data structures can be described as listed below.

3.3.1 ArrayList

ArrayList uses a dynamic array internally in the implementation. This implementation is not synchronized. It can be initialized when defined, and the default capacity is 10 while the default load factor is 0.75. ArrayList increments 50% of the current size when the number of elements exceeds its threshold value and which we call the growing policy of the ArrayList. For example, if we define an ArrayList with an initial capacity of 10, and the number of elements reaches 7, the total capacity increases to 15. Its constructer is defined as *ArrayList(int initialCapacity)* shows that it can be initialized with an integer value.

3.3.2 LinkedList

LinkedList internally uses a doubly linked list to store data, and this implementation is also not synchronized. When the number of elements exceeds its capacity, it gets incremented by 50% of the current size. LinkedList neither has an initial capacity nor does it have a load factor implemented to its constructor.

3.3.3 Vector

Vector is a synchronized data structure. As its growing policy describes, it increments 100%(doubles the capacity) of the current size when the number of elements reaches its threshold value. The capacity of a Vector is at least as large as the vector size. Its constructor has the initial capacity parameter, and also we can define a load factor to it as well(*Vector(int initialCapacity, int capacityIncrement)*).

3.3.4 HashMap

HashMap is a non-synchronized implementation with a default capacity of 16. The default load factor is 0.75. Creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table. The default threshold is 16 × 0.75 = 12, which means that it will increase the capacity from 16 to 32 after the 12th entry (key-value-pair) is added. This also means that HashMap doubles its capacity when it reaches its threshold. The other thing to note is that HashMap doesn't guarantee how the elements are arranged in the Map.

3.3.5 TreeMap

By default, TreeMap sorts all its entries according to their natural ordering. For an integer, this would mean ascending order and alphabetical order for strings. We could also define custom order as well. The implementation is not synchronized. TreeMap does not have an initial capacity or a loading factor implemented. That is why TreeMap does not reallocate nodes on

adding new ones. Thus, the size of the TreeMap dynamically increases if needed, without shuffling the internals. So it is meaningless to set the initial size of the TreeMap.

3.3.6 LinkedHashMap

LinkedHashMap is almost similar to the HashTable, though the difference is that it uses a linked list internally to enhance the functionality of the hash map. It maintains a doubly-linked list running through all its entries in addition to an underlying array of default size 16.

Even though six major data structures are listed above, more versions can be derived. The Map interface has HashMap, LinkedHashMap, and TreeMap, and for the List interface, it's LinkedList, ArrayList, and Vector. But, since HashMap, LinkedHashMap, ArrayList, and Vector can have the initial capacity when they are initialized, we could derive four more data structures as the initialized versions of those data structures. Now all together, we have ten data structures for the analysis.

Table 3.1: Summary of the selected data structures.

Data Structure	Features
ArrayList	• Load factor = 0.75 • Increment = 50% • Default capacity = 10
LinkedList	• Increment = 50%
Vector	• Increment = 100%
HashMap	• Load factor = 0.75 • Default capacity = 16
TreeMap	• Sorted • Not synchronized
LinkedHashMap	• Default capacity = 16 • Uses a LinkedList internally

3.4 Importance of Run-time Behaviour Data

Every data structure operation takes the memory and the CPU run-time when it gets executed. For example, if we consider HashMap, the process of rehashing takes place every time it dynamically gets increased. This rehashing process takes CPU run-time in a noticeable amount every time. Also, when the capacity is increased, it causes to take up a noticeable amount of memory consumption to keep those extra buckets. Altogether, both the run-time and the memory consumption affect overall performance and the efficiency of the programs. For smaller programs, in the sense of programs that handle not very larger amount of data, this performance difference might not be visible. But, performance matters a lot when it comes to large-scale, multi-tiered enterprise-level applications.

Having measured the run-time and the memory usage of a data structure in a particular program, we can get information like how it has increased dynamically, how much memory has cost throughout the execution, how the growing policy has been affected, etc. But, by having data for just one data structure, we cannot differentiate between other data structures. So, we need to collect data on the run-time behaviour of all the data structures that we want to analyse.

3.5 Ways of Measuring the Run-time and the Memory Usage

First of all, there are several methods of measuring both the run-time and the memory consumption of a Java program. Basically, since we are trying to measure how a particular data structure would have cost memory and the run-time in a Java program, we need to address the smallest part of the process execution to achieve better results.

3.5.1 Getting the Run-time of a Certain Operation

A Java built-in function named *System.nanoTime()* has been used to measure the run-time of a particular process in a program. This function gives extremely precise time in nanoseconds relative to some arbitrary point. The following code snippet shows how we can use this function to measure the time taken by a process.

```
final int ITERATIONS = 1000;
final int AVERAGE_NUMBER = 3;

List<Integer> list;

long startTime = System.nanoTime();
for(int i=0; i<AVERAGE_NUMBER; i++) {
  list = new ArrayList<>();
  for(int j=0; j<ITERATIONS; j++) {
    list.add(j);
  }
}
long endTime = System.nanoTime();

long totalTime = (endTime – startTime)/AVERAGE_NUMBER;
```

In the above example, *totalTime* would give the time taken by all the elements to be added to the *list* in nanoseconds.

3.5.2 Getting the Memory Usage of a Certain Operation

Java Runtime class provides us access to the Java garbage collector. We can run the garbage collector and calculate the memory usage of a process in the run-time. The following code snippet shows an example of how it can be done.

```
final int ITERATIONS = 100000;
final int AVERAGE_NUMBER = 3;
List<Person> list = new ArrayList<Person>();

for(i=0; i<AVERAGE_NUMBER; i++) {
      for (int j=0; j<=ITERATIONS; j++) {
            list.add(new Person("Tom", "Male"));
      }
}

Runtime runtime = Runtime.getRuntime();
runtime.gc();
long memory = (runtime.totalMemory() –
      runtime.freeMemory())/AVERAGE_NUMBER;
```

In the above example, the variable *memory* gives the memory usage of the process that happens inside the for loop, in this case, adding elements to an ArrayList. Here we can access the Java

garbage collector using the *runtime* variable and hence run it to get the total memory and the free memory. After that, the memory usage can be calculated as shown.

Though we could calculate memory usage like this, it should be noted that the result is not 100% accurate. No matter how we try, we cannot get the exact memory usage. There are tools like Classmexer [14] that use Java instrumentation [15] inside the implementation to get the memory of an object reference, including subobjects(objects referred to by a given object). But it also does not give the exact memory usage as it is impossible to achieve that precision(after using those tools, it was found that all those methods give the same results). The other thing is that we get the same memory usage for all the relevant data structure operations. That means, here, the *insert()* function is the only considerable function in terms of memory calculation because it makes sense that the insert() operation costs memory. Even though it might seem like we could measure the memory, for example, in the delete() operation, the reduced memory when the elements are deleted is not achievable with the above methods of calculating memory.

We can use both methods in any Java program to measure either run-time or memory usage of any execution of the particular program. First, we have to have an idea about where these methods should be used accordingly. Here, in this case, the calculations should be done to the execution processes of each data structure. For example, as it is shown in the above code snippets, for a data structure, execution operations are adding elements, searching elements, deleting elements, etc. So, in order to analyse the run-time behaviour of each data structure process, we need to calculate run-time and the memory usage for each process individually for each selected data structure.

Even though all the processes can be executed by using a single function and getting the total run-time and the memory usage for each data structure individually, depending on the use case, every data structure differs from the run-time behaviour for each process operation(*add()*, *contains()*, etc.) they execute. For example, a particular data structure might be better at storing and accessing data, while the other data structure is better at searching for elements inside the data structure. That totally depends on the implementation of the data structure and the developer's use case. That is why the runtime behaviour data should be collected individually for each operation.

3.6 Methods of Collecting Run-time Behaviour Data in Java Programs

Two approaches have been tested out in this particular research work to collect data about the run-time behaviour of selected data structures.

3.6.1 Testing Environment

The following list contains the developer environment for all the testing related to the research work.

- Java 11.0.12
- Java(TM) SE Runtime Environment 18.9
- Java HotSpot(TM) Server VM 18.9
- Intellij Idea
- Intel 1165G7 Core i7
- 20GB RAM
- 512GB SSD

3.6.2 Method 1: Using Java Programs from Github to Collect Data

Github is a code hosting platform for version control and collaboration where millions of developers contribute to the open-source community. Github contains tons of programs developed by developers around the world, pretty much in every programming language, including Java.

So, this being the first method, a large-scale dataset [13] of a list of Github repositories was collected and processed to find the ones that could be built in the above environment. After that, all the Java programs with test cases were selected through a web crawler written in python [16]. After collecting more than 100 programs to run the tests, with the help of the Javassist library [11], those programs were analysed to search for the usage of data structures in the implementation of each program. Then, using the mentioned methods of collecting run-time and

the memory usage in the previous section, a data set of the run-time behaviour of a reasonable amount of programs was collected.

Yet, after having analysed the collected data, it was realized that those data do not show much of a performance gap between each operation in terms of either run-time or memory usage. Furthermore, the way those programs have used data structures to do certain processes, any of those processes do not handle enough data to show significant performance gaps between operations. The basic reason is that we cannot find any multi-tiered, large-scale, enterprise-level programs that handle a large amount of data with data structures on Github. Even if those programs exist, developers do not specifically reveal their source codes because those programs are production level programs.

So, even though this method is a perfect method for collecting such data, in this case, it does not fit with the research for the above reason. That is why we have to use an alternative approach to address this problem.

3.6.3 Method 2: Implementing a Program Solution to Benchmark Each Data Structure

In this method, a program solution written in Java where that basic implementation can be found with the link [17], was used to collect the run-time behaviour data for each selected data structure (the link to the executable jar file that has been used to collect final data can be found with this link [18]).

According to Marten Vooberg's method [12] of measuring the run-time and the memory usage of a data structure, first, we need to generate a random number of data specifically, in this case, data being integers and string values. Figure 3.2 shows an illustration of the basic steps of the method. The method starts generating data as the inputs of all the data structures used for the analysis.

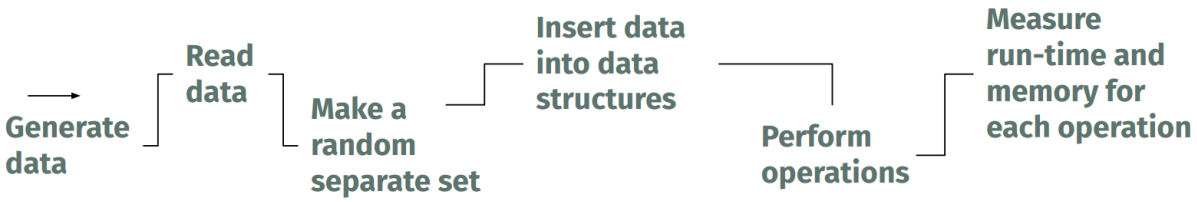

Figure 3.2: The benchmarking process.

- **Generate data**

A random set of integer values has been generated using a Python script(which can be found on the same repository where the benchmarking program is). To be precise, 2^{20} integer values are generated through the program. All the data values are unique and uniformly distributed. Along with these integer values, randomly generated individual UUID strings are used to put in Map data structures.

- **Read data**

In this step, generated integer values are read by the implemented benchmarking program solution. After that, all the integers are saved inside an ArrayList(for the List data structures) and a HashMap(for the Map data structures) for further use. When we save values in the HashMap, a random and unique UUID string is generated within the program and stored as the key for each integer value.

- **Make a random, separate set**

Along with the previous step, another random set from the read data is taken out with the help of an algorithm called the reservoir algorithm. This separate sample data set is used to perform various operations like searching elements, deleting elements, etc., inside each data structure.

- **Insert into data structures**

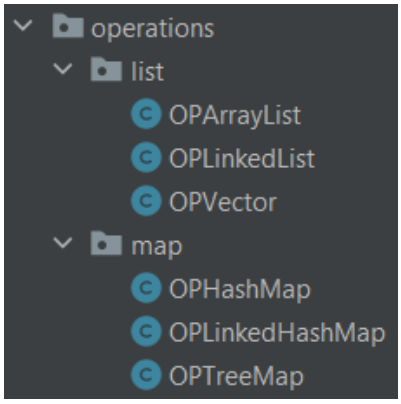

Figure 3.3: Class Structure of the data structures that are used to perform operations.

As shown in Figure 3.3 above, classes have been defined to store each data structure. Inside each class, there is an object instance of the relevant data structure. In a single run of the benchmarking process, we only collect data for data structures either from the List interface or the Map interface. Since the operations inside each interface are different, there can be conflicts(confusions) between labeling and output data. Therefore it cannot be run for both interfaces in a single run.

In the beginning, inside each class, an empty data structure is initialized to perform each and every operation accordingly.

- **Perform operations**

Each class in the above figure has a set of methods implemented that include each individual data structure operation. An example method is given in the simple code snippet next page.

```
public void insertValues() {
      for (Integer val:getListData()) {
            linkedList.add(val);
      }
}
```

So, we have a method like this for each operation to perform the necessary tests.

- **Measure run-time and memory usage for each operation**

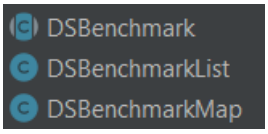

Figure 3.4: Class structure of benchmarking.

As shown in Figure 3.4, the benchmarking part is handled by two classes extended to an abstract class named *DSBenchmark*. *DSBenchmarkingList* handles the List data structures while *DSBenchmarkMap* handles the benchmarking of Map data structures.

Inside each class, there are two methods to calculate the run-time and the memory usage of each data structure operation separately. They work just like the code snippets shown under section 3.5.

There are a few input parameters for the benchmarking program to run the tests. Those are trials, starting capacity, increment value, and the number of increment steps.

1. **-list** or **-map**: -list means data structures from the List interface, and -map means data structures from the Map interface. The program will calculate only one of these at each run.
2. **-r** or **-m**: -r means run-time, and -m means memory usage. The program will calculate only one of these at each run.
3. **Trials**: Number of iterations that each process should run (higher the iteration, lower the noise in the calculations, but it takes time).
4. **Starting Capacity**: Starting capacity of each data structure.
5. **Increment value**: Number of integers that get added to the list at each step.
6. **Number of increment steps**: Number of steps should the increment value that gets added to the capacity.

After having a reasonable number of iterations of benchmarks, by using the trial and error method, it was realized that the ideal values for the starting capacity and the increment value are 100,000 and 50,000, respectively. The trial was taken as 10, which means one operation would run ten times and return the average value of the calculations to reduce the noise in the collected data. Altogether, with the jar file created, the benchmark was run for both the set of data structures in the sense List and Map, each 100 steps to collect data for further analysis. An example command is shown below.

```
java -jar benchmark.jar -list -r 10 100000 50000 20 runtime_list.csv
```

This example shows that if we run the program for data structures from the List interface, It would run all the common operations discussed under section 3.3 for each data structure(ArrayList, LinkedList, and Vector) in a single. This particular case will collect run-times for the executions, and after all the iterations, the program will produce a CSV file with the collected data.

3.7 Building a Machine Learning Model with Collected Data

Now that we have collected a fair amount of data for each data structure, the next step is to build a machine learning model to identify patterns and make the necessary predictions to suggest data structures depending on the need.

3.7.1 Development Environment

The following list contains the development environment, including all the packages that were used to develop the machine learning model.

1. Python 3.8.12
2. Pandas
3. Sklearn
4. Xgboost

3.7.2 Preprocessing the Data

By looking at the collected data of both run-time and memory usage, it was identified that the most suitable machine learning model would be a multiclass classification model. The reason is that each data row belongs to a specific class, that is, the data structure(note that run-time is in milliseconds and the memory usage is in megabytes).

Table 3.2: Snapshot from the dataset(run-time in milliseconds).

Number of Elements	Data Structure	insert()	insertAt()	contains()	indexOf()	get()	delete()
100000	ArrayList	6.75	10.76	3.08	3.13	0.11	10.86
100000	ArrayList_Initialized	2.94	5.23	0.28	0.26	0.07	5.09
100000	LinkedList	8.39	1.57	2.80	0.84	1.50	0.17
100000	Vector	7.39	10.81	2.67	3.13	0.16	14.04

As shown in Table 3.2, the whole dataset for either the Map interface or the List interface can be classified into the labelling of those particular data structures. In this example, it's ArrayList, LinkedList, and Vector. Since more than one class(here, it has three classes) is there, the best fitting model will be a multiclass classification model.

Collected data can be separated into six different sets. Because we can consider each operation(*insert(), insertAt(),* etc.) individually. The reason is that the developer does not use all the operations same time, and this way, we could present it in a more understandable way. For a program to be efficient, we need to reduce the run-time and the memory usage. When it comes to data structures, in terms of run-time or memory usage, for a particular operation, the data structure that has consumed minimum run-time or memory usage can be considered as the most suitable data structure to perform that operation. So, for example, let's take the *insert()* operation. By looking at Table 3.3, we can see that number of inserting elements into each data structure is 100000, and for each data structure, we have calculated the run-time. So, for the *insert()* operation, the minimum runtime value is 2.94, which belongs to the initialized version of the ArrayList. That means, out of the four data structures, to perform the *insert()* operation to

100000 elements, the initialized version of the ArrayList takes only 2.94 milliseconds. That tells us that it is the most suitable data structure among the other four to handle the situation. Like this, the number of elements increases, and for each set, we have calculated the run-time just like in Table 3.3. On the other hand, the dataset can be grouped by the number of elements here; it's *NumberOfElements*. So, we get a total of six parts of the whole data set, considering there are six operations.

Table 3.3: Two parts of the split sets from the original dataset.

NumberOf Elements	DataStructure	Insert()
100000	ArrayList	6.75
100000	ArrayList_Initialized	2.94
100000	LinkedList	8.39
100000	Vector	7.39
100000	Vector_Initialized	2.96

NumberOf Elements	DataStructure	InsertAt()
100000	ArrayList	10.76
100000	ArrayList_Initialized	5.23
100000	LinkedList	1.57
100000	Vector	10.81
100000	Vector_Initialized	5.18

After considering the minimum value of the operation column, we can create a new column as the recommended data structure for each operation. In Table 3.4, it is shown that for the first group, the suitable data structure would be *ArrayList_Initialized*, and for the next group, it would be *LinkedList* since it has the minimum run-time for the *insert()* operation.

Table 3.4: New column with the recommended data structure for each column.

NumberOfElements	DataStructure	insert()	ds_insert()
100000	ArrayList	6.75	ArrayList_Initialized
100000	ArrayList_Initialized	2.94	ArrayList_Initialized
100000	LinkedList	8.39	ArrayList_Initialized
100000	Vector	7.39	ArrayList_Initialized
100000	Vector_Initialized	2.96	ArrayList_Initialized
…	…	…	…
10050000	ArrayList	17.17	LinkedList
10050000	ArrayList_Initialized	14.21	LinkedList
10050000	LinkedList	6.86	LinkedList
10050000	Vector	10.05	LinkedList
10050000	Vector_Initialized	8.65	LinkedList

Just like explained, for each operation, it can be decided which data structure is better, and the same can be done when deciding on the memory usage as well. Also, we can use the same process to decide between Map data structures, and after all that, we are ready to create the classification models to make suggestions for each operation at a particular number of elements that the data structure will process.

So, this was the overall method used to get the most accountable results we wanted, for which to have a model that could suggest data structures depending on the user's needs. Moreover, the results and the outcomes of the whole process will be discussed in the next chapter.

CHAPTER 4
RESULTS AND DISCUSSION

This chapter highlights the results of the study carried out throughout the research. Each part is discussed with necessary snapshots included in the discussion. The main results discussed are the results of run-time behaviour and the results from the machine learning model.

4.1 Information Provided by the Collected Data

After analysing the data both for run-time and memory usage, data can be visualized for data structure operations and see how they behave. Also, get an idea about the input parameters of the classification model in order to achieve better results.

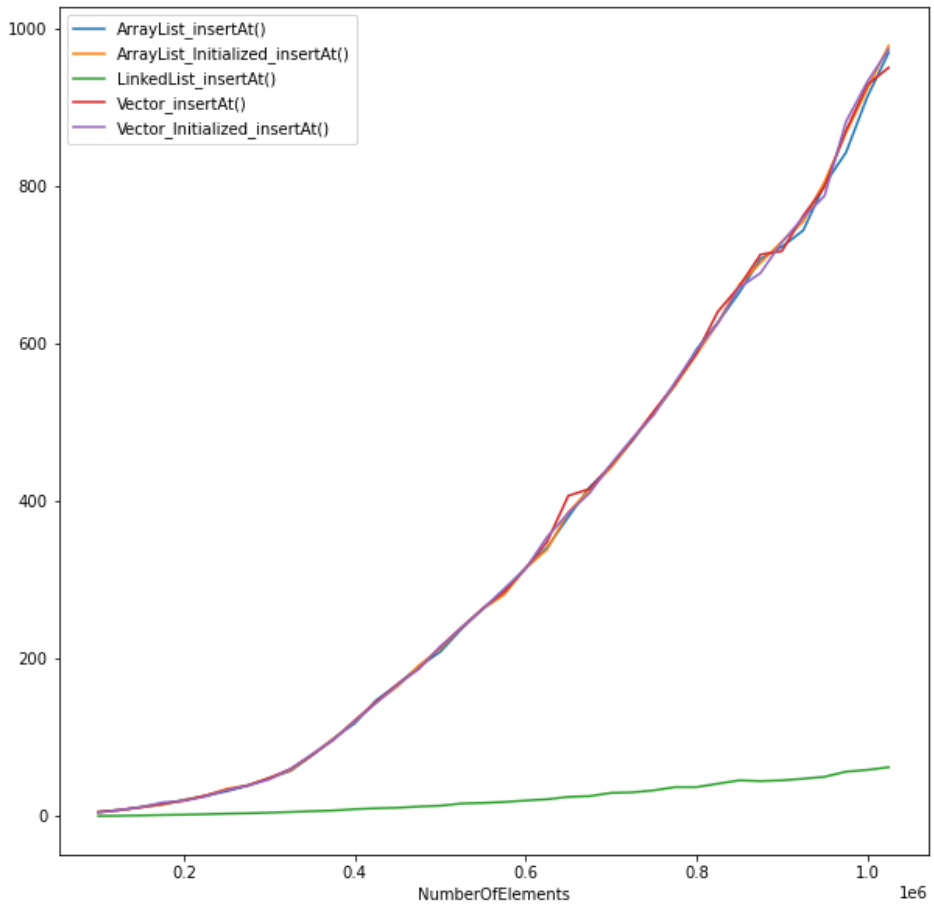

Figure 4.1: Run-time(ms) behaviour of *insertAt()* operation of each data structure from the List interface.

For example, see Figure 4.1 above(the x-axis shows the number of elements inserted at each step). We can see that relative to the LinkedList(green colour line), other data structures have consumed more run-time to perform the particular *insertAt()* operation. That means LinkedList takes less time that other data structures in the List interface to perform *insertAt()* operation of a set of data. On the other hand, we could recommend LinkedList to have in a program if it has more *insertAt()* operations performing on the processing data.

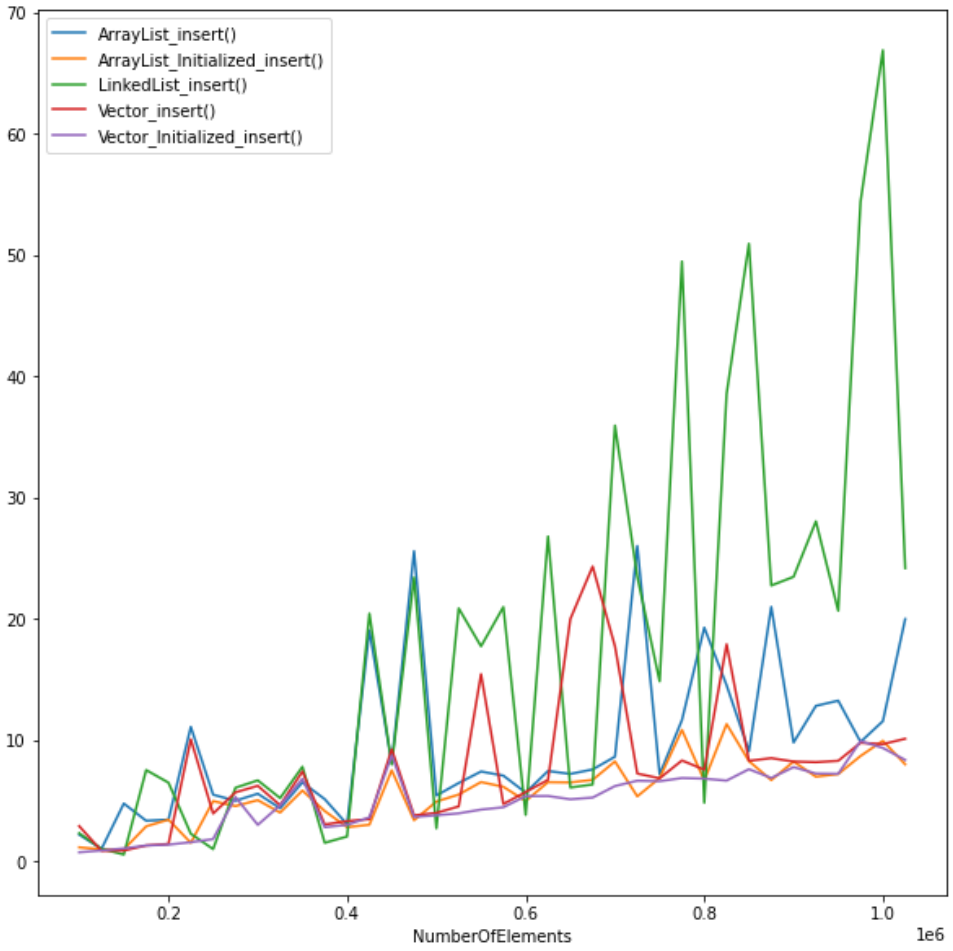

Figure 4.2: Run-time(ms) behaviour of *insert()* operation of each data structure from the List interface.

Similarly, if we look at Figure 4.2, to perform the *insert()* operation, LinkedList takes much time. Instead, we could suggest the initialized versions of either ArrayList or Vector because they both consume the run-time similarly for the *insert()* operation.

If we take data structures from the Map interface, Figure 4.3 below shows how the *put()* operation behaves in run-time. We can see that the TreeMap consumes way more runtime, becoming noticeably the slowest one among the rest. Also, the clear difference between the initialized versions of HashMap and LinkedHashMap and the non-initialized version is visible. The initialized version of HashMap takes the lowest time to complete the operation than the rest.

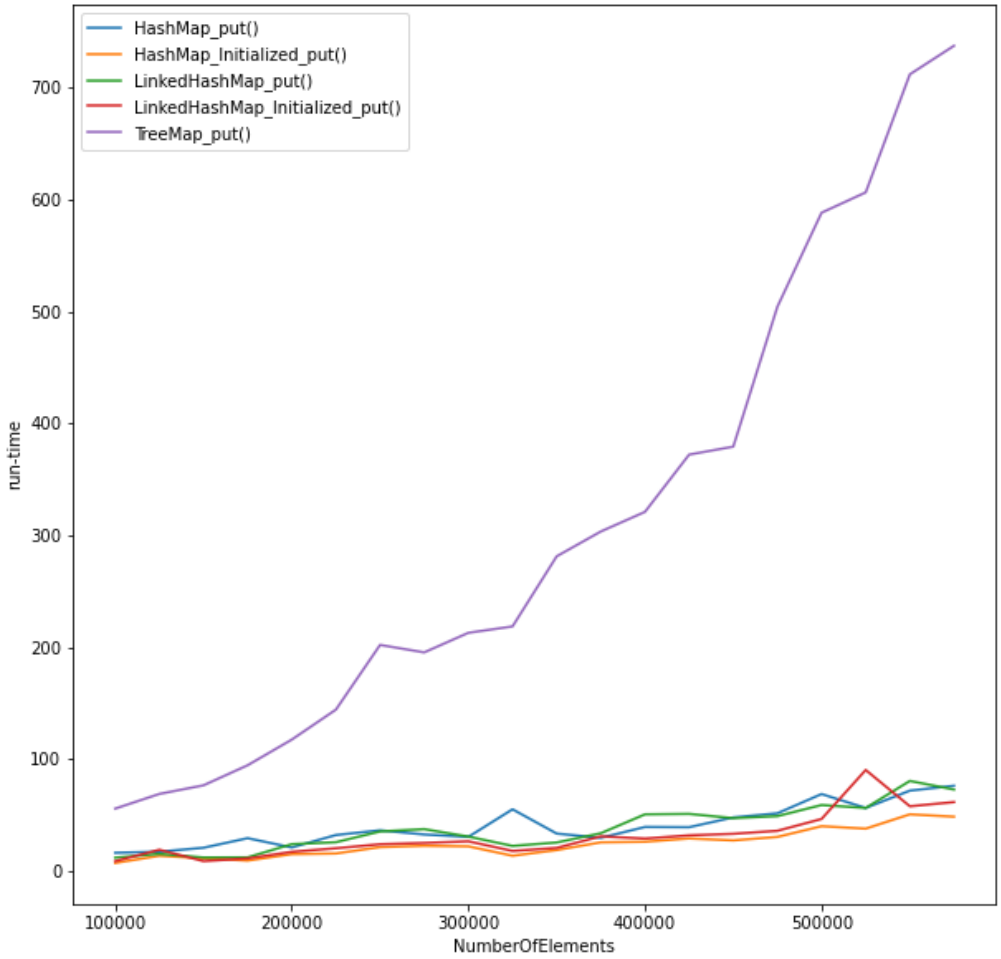

Figure 4.3: Run-time(ms) behaviour of the *put()* operation in each Map data structure.

When it comes to searching for a value inside a Map, Figure 4.4 shows how all the Map data structures behave. As usual, TreeMap takes a lot of time to complete the operation. LinkedHashMap and the initialized version of it perform the same way, also achieving the lowest run-time. Looking at both the graphs, we can say that TreeMap is not recommended as it consumes way more run-time to perform a particular operation.

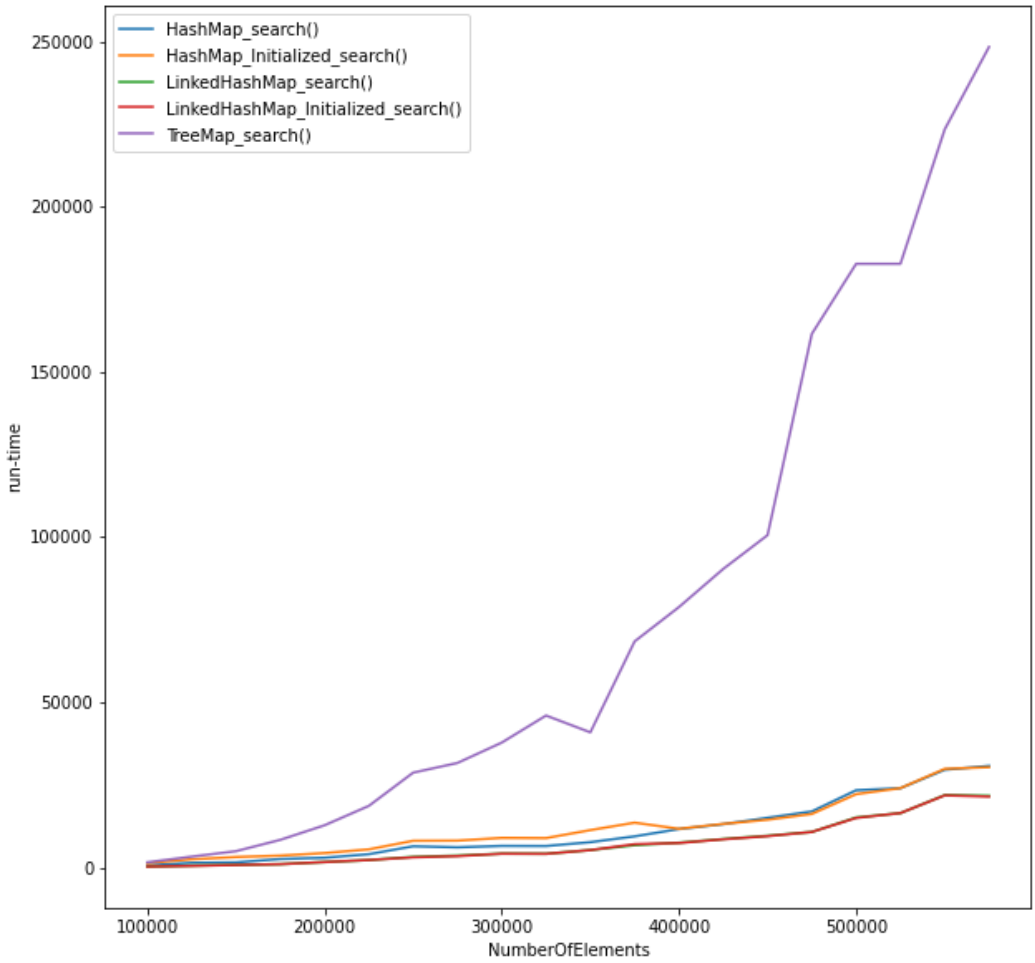

Figure 4.4: Run-time(ms) behaviour of the *containsValue()*(search) operation in each Map data structure.

Figure 4.5 below shows how the memory consumption has been throughout all the steps for the List data structures. As we can see, the Vector and the ArrayList from the red colour and the blue colour lines, respectively, they have consumes less memory than the rest of the data structures. Even when the number of elements gets higher, the ArrayList has started to use less memory than all the other data structures. The initialized version of the Vector(purple colour line) has used more memory because when it is initialized with a value, a certain amount of memory is needed to keep those extra buckets. In some cases, even though this reduces the processing power by reducing the run-time since it does not have to increase its size dynamically, it does have to use more memory to keep up with that. The LinkedList(green color

line) has taken the most of the memory as it uses a doubly linked list inside the implementation. Apparently, for all the operations, the memory consumption is the same.

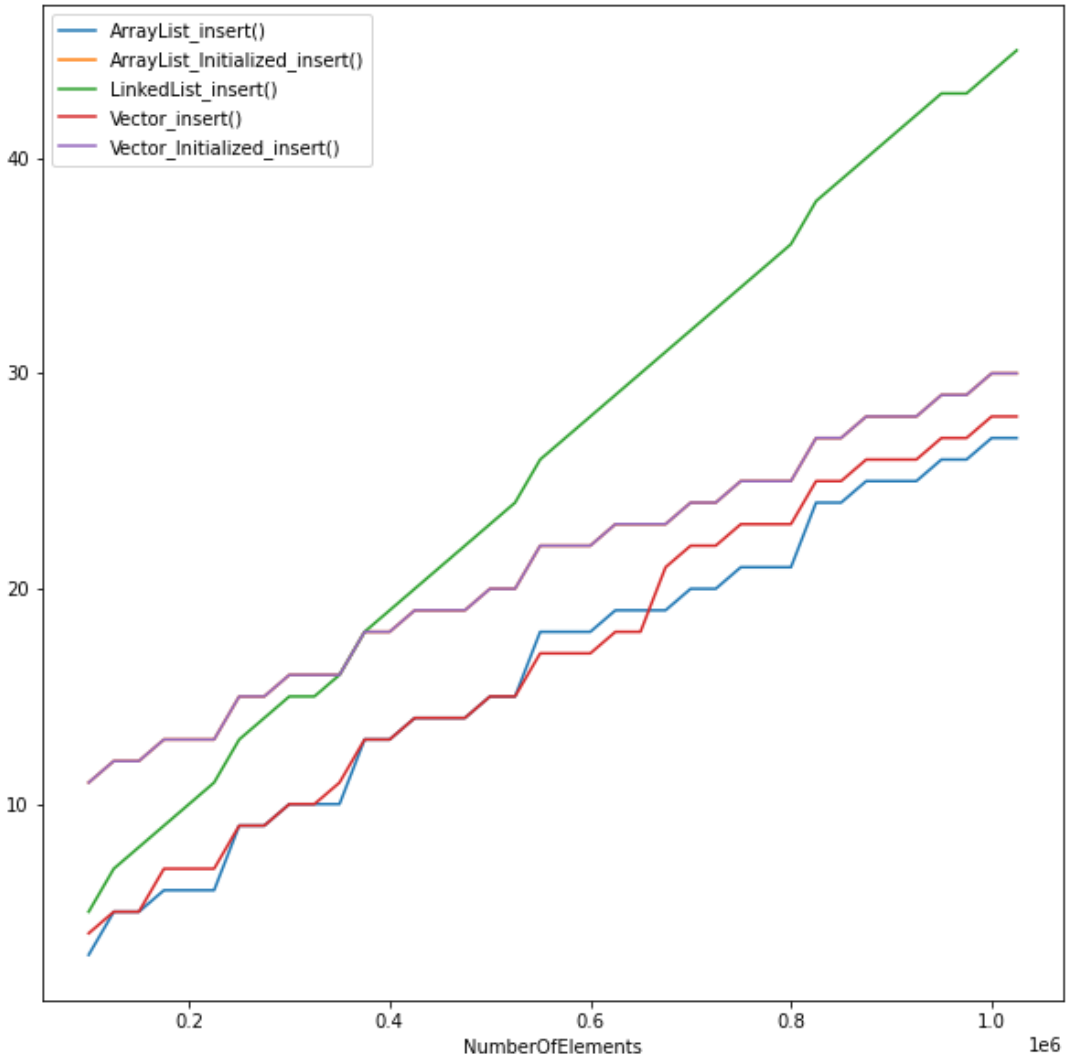

Figure 4.5: Memory usage(MB) of each data structure from the List interface for the insert() operation.

Figure 4.6 below shows how the memory usage varies in each data structure for the Map interface. Here as the keys, string values have been used and integer values as the values.

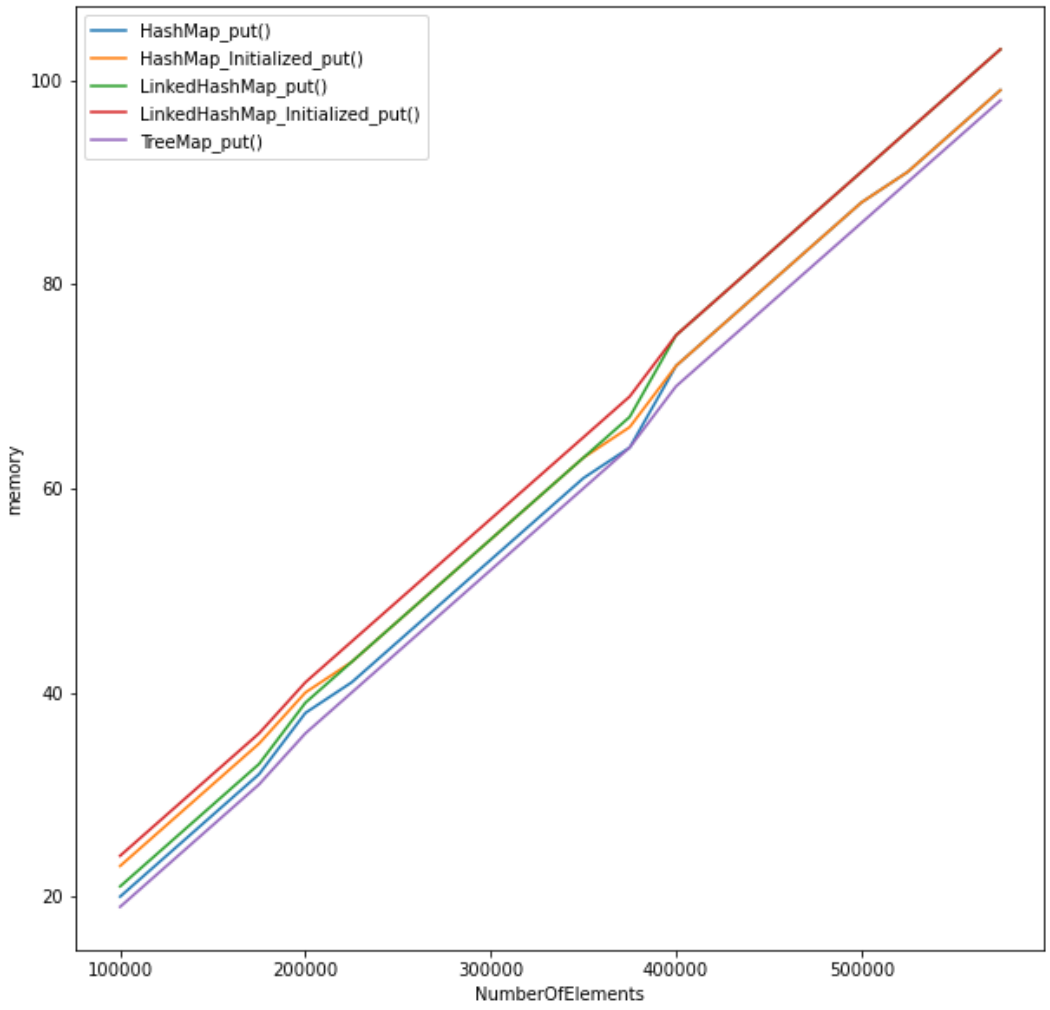

Figure 4.6: Memory usage(MB) for the Map data structures.

String values are generated using a UUID generator and put into an initial map data structure(HashMap) with the integer values for further processing. We can see that TreeMap has consumed the lowest memory of all in the graph. Also, considering other Map data structures in the graph, we can see that the initialized versions of them have consumed memory similarly after some point. The rehashing happens in the regular map data structure(HashMap or LinkedHashMap), and it might have increased its size to the same as it is in the initialized version. So, we can use this information to differentiate and determine how the maps should be used depending on the developer's need.

All this information helps to understand the use cases or the scenarios that those data structures fit the best depending on various tasks. But, just by having these graphs, we cannot still suggest

any data structure for a particular use case. We need a better classification algorithm to classify all the data from the datasets gathered. So, after having leant the patterns of the run-time behaviour of each data structure, we could only suggest any data structure for the developer. The above information helps a lot to achieve that task with good precision.

4.2 Choosing a Machine Learning Model

It does not matter if the data structure is from the Map interface or the List interface; each has six common operations. But, since each interface works differently, we have to separate the two interfaces and consider each separately. Also, each operation has its own unique run-time behaviour in terms of both run-time and memory usage. We can see that in the collected data as well. Also another reason is that the developer does not use all the operations at once. The developer might use a data structure only to store and retrieve elements. So, we have to focus only on those two operations for a case like that. Therefore, we need to take all the six operations individually for each interface and analyse them accordingly to get the output we want.

As explained in section 3.3, we have together ten data structures for the analysis. Five from the Map interface and the other half from the List interface. This means we could consider these data structures as our classes for the chosen model. So, five classes from the Map interface and five from the List interface. Therefore a multiclass classification model fits the best for the scenario.

We can fit a classification model for each data structure operation and see how the algorithm catches the pattern. But, for some operations; for example, at each iteration, LinkedList is the one that consumes the minimum run-time of all for the *insertAt()* operation. We can see it in Figure 4.6 below.

4.3 Analysing Data Further

Even though we could fit a suitable machine learning model for each operation, it is better to analyse the collected data further in terms of both run-time and memory usage. The reason is

that there might be cases where fitting a machine learning model could be a waste because there can be visible situations that show which data structures are the most suitable ones.

4.3.1 Analysing the Run-time

We can fit a classification model for each data structure operation and see how the algorithm catches the pattern. But, for some operations; for example, at each iteration, LinkedList is the one that consumes the minimum run-time of all for the *insertAt()* and the *detete()* operations. We can see it in Table 4.1 and Table 4.2 below.

Table 4.1: Sample dataset; LinkedList has the minimum run-time(ms) for the *insertAt()* operation.

NumberOfElements	DataStructure	insertAt()	ds_insertAt()
100000	ArrayList	10.76	LinkedList
100000	ArrayList_Initialized	5.23	LinkedList
100000	LinkedList	1.57	LinkedList
100000	Vector	10.81	LinkedList
100000	Vector_Initialized	5.18	LinkedList
150000	ArrayList	12.7	LinkedList
150000	ArrayList_Initialized	12.73	LinkedList
150000	LinkedList	1.5	LinkedList
150000	Vector	11.94	LinkedList
150000	Vector_Initialized	12.37	LinkedList
200000	ArrayList	20.61	LinkedList
200000	ArrayList_Initialized	20.08	LinkedList
200000	LinkedList	2.31	LinkedList
200000	Vector	20.17	LinkedList
200000	Vector_Initialized	20.25	LinkedList
250000	ArrayList	31.32	LinkedList
250000	ArrayList_Initialized	31.88	LinkedList
250000	LinkedList	3.95	LinkedList
250000	Vector	31.44	LinkedList
250000	Vector_Initialized	32.18	LinkedList

Table 4.2: Sample dataset; LinkedList has the minimum run-time(ms) for the *delete()* operation.

NumberOfElements	DataStructure	delete()	ds_delete()
100000	ArrayList	10.86	LinkedList
100000	ArrayList_Initialized	5.09	LinkedList
100000	LinkedList	0.17	LinkedList
100000	Vector	14.04	LinkedList
100000	Vector_Initialized	5.1	LinkedList
150000	ArrayList	12.77	LinkedList
150000	ArrayList_Initialized	12.76	LinkedList
150000	LinkedList	0.11	LinkedList
150000	Vector	11.99	LinkedList
150000	Vector_Initialized	11.47	LinkedList
200000	ArrayList	20.31	LinkedList
200000	ArrayList_Initialized	20.26	LinkedList
200000	LinkedList	0.13	LinkedList
200000	Vector	20.49	LinkedList
200000	Vector_Initialized	20.23	LinkedList
250000	ArrayList	31.52	LinkedList
250000	ArrayList_Initialized	31.82	LinkedList
250000	LinkedList	0.15	LinkedList
250000	Vector	32.42	LinkedList
250000	Vector_Initialized	32.15	LinkedList

The tables show the run-time taken at each step by every data structure in the List interface. *insertAt()* and *delete()* columns show the run-time and the *ds_insertAt()* and *ds_delete()* columns show the data structure with the lowest run-time which is the most suitable data structure among the rest. Hence, we could derive that using a LinkedList for *insertAt()* and *delete()* operations would be the most suitable way.

According to the processed data, it was able to see that for the *containsValue()* operation, both the initialized and the non-initialized versions of LinkedHashMap consume the minimum run-times. It is evident that the initialized version takes, even more, less time to complete the operation because it does not have to rehash itself when the elements are added. But depending on the situation, both versions can be used wisely. That decision totally depends on the use case and the developer. So we could easily not classify the dataset for the *containsValue()* operation.

4.3.2 Analysing the Memory

As mentioned in section 3.7.2, we take the data structure that costs minimum memory usage as the deciding factor of the most suitable data structure for the amount of data processed. Also, as mentioned in section 3.5.2, we are only able to consider the input or the insert operation of each data structure since all the operations give the same memory usage results. Straight away, we can neglect the initialized version of the chosen data structures as they keep empty buckets ahead of the data structure to accept the inserting elements and to save the run-time. So, looking at the graph in Figure 4.5, we can see that LinkedList has consumed way more memory because it used a doubly linked list in the implementation. Each node keeps a value as well as a reference to the next node. ArrayList and Vector cost less memory, having similar lines along the graph. Collected data shows that Vector costs less memory than ArrayList in the range of 550000-650000 number of elements. But, overall, ArrayList costs less memory than the rest of the List data structures. So, the developer can be motivated to use either an ArrayList or a Vector depending on the need to cost less memory in their programs. It is also recommended to compare the run times of both data structures.

The graph in Figure 4.6 shows that TreeMap consumes the minimum memory of the rest of the Map data structures. But, as explained in the run-time section 4.3.1 above, TreeMap is the worst performing data structure in terms of run-time. Both Figure 4.3 and Figure 4.4 have evidence for that. So, the next data structure that costs the lowest memory is HashMap. HashMap maintains that throughout all the iterations of the number of elements. So, using any classification model is not worth it because we can see the results by analysing the data set.

4.4 Classification Results

As it is mentioned in the previous sections, memory usage data does not need to be classified since the processed data clearly shows the results we need. On the other hand, run-time data can be classified as opposed to the mentioned points in section 4.3.1. Considering each data structure operation separately, the classification results of run-time data that are worth classifying for the data structures from both List and Map interfaces are discussed in this section.

4.4.1 Results for the List Data Structures

For the *insertAt()*(replacing operation) and the *delete()* operations, LinkedList is the only data structure that has the lowest run-time of all. So, those operations need not be classified. Rest of the four operations, data can be classified using a multiclass classifier. Here, in this case, the Xgboost classifier has been used for the classification process. First, the data set was divided into two parts; the test set being 70% of the whole data set and the train set being 30% of the original data set. The train set was used to fit the classification model, and the test set was used to validate the built classification model. The precision results are as in Table 4.3 below.

Table 4.3: Classification results of the data structure operations from the List interface

Class	insert()	contains()	indexOf()	get()
ArrayList	0.8333	0.9044	0.8707	0.9873
ArrayList_Initialized	0.7583	0.8980	0.8268	1.0000
LinkedList	0.8528	-	-	-
Vector	0.7000	0.9643	0.9231	0.8333
Vector_Initialized	0.7419	0.8250	0.8333	0.4000
Precision	0.80	0.90	0.85	0.98

4.4.2 Results for the Map Data Structures

Like the data structures from the List interface, the data set for the Map interface data structures were divided into two parts as well; 70% for the test set and 30% for the train set. Classification results are shown in Table 4.4 below. Treemap is not in the resultant classes because, as Figure 4.3 and Table 4.4 show, it takes enormous time to complete an operation compared to other data structures.

Table 4.4: Classification results of the data structure operations from the Map interface

Class	put()	replace()	containsKey()	get()	remove()
HashMap	0.5000	0.8333	0.8065	0.9808	0.8889
HashMap_Initialized	0.9485	0.8723	0.9149	0.8936	0.8933

LinkedHashMap	0.7692	0.9444	0.9412	1.0000	-
LinkedHashMap_Initialized	-	0.8095	0.9524	1.0000	1.0000
Precision	0.91	0.86	0.89	0.94	0.90

CHAPTER 5
CONCLUSIONS

Analysing the selected optimized Java data structures and suggesting better data structures that fit into the developer use case is the primary goal of this research work. The usage of all the research work is mainly focused on large-scale Java enterprise applications since they heavily use data structures. After analysing the collected run-time and memory usage data, we have concluded the methods of recommending suitable data structures for suitable use cases. In terms of memory consumption, we can use either an ArrayList or a Vector from the List interface, depending on the situation, to achieve the minimum of memory usage. Even though the TreeMap uses less memory overall, because of how bad it behaves in the run-time section, instead of it, we could use HashMap to achieve less memory from the Map interface.

The classification was only needed when analyzing the run-time data for certain operations. For example, LinkedList in the List interface costs the lowest run-time in terms of *insertAt()* and *delete()* operations. So we do not need to apply any classification for cases like this. However, the data worth classifying were classified, and the results are shown under the results and discussion section accordingly. Hence the derived classification models can suggest data structures for all the operations individually. So, we could use these analysed and classified results to propose or suggest data structures for a particular use case, addressing the third objective of the research to propose a system capable of generating an assisting report.

CHAPTER 6
FUTURE WORK

This research work can further be extended to design a Java library that uses all the results and classified models to decide between data structures that suit the best for Java user programs. This would even help Java developers a lot to identify data structures and use them wisely in their programs to make them run efficiently in any system. To achieve this, we need to collect some real-world, large-scale applications that use data structures to handle a large amount of data. We can benchmark those programs and use run-time and memory usage information to build more convenient and suitable applications for real-world use. By collecting more and more data after running many amounts of iterations, we can make our classification models more robust and precise very well.

Even though this research work only focuses on Map and List interfaces, there are a few other optimised in built Java data structures as well. So, to build a complete solution, those data structures(for example, data structures like Stacks, Queues, etc.) can be considered and benchmarked to make necessary suggestions where appropriate.

REFERENCES

[1]. Woodside, M., Franks, G., Petriu, D.C.: The future of software performance engineering. In: Future of Software Engineering (FOSE) (2007)

[2]. Wu, X., Woodside, M.: Performance modeling from software components. SIGSOFT Softw. Eng. Notes 29(1), pp. 290–301 (2004)

[3]. M. Harkema, D. Quaetel, B.M.M. Gijsen, R.D. van der Mel: Performance Monitoring of Java Applications (2002)

[4]. Wenlong Li, Eric Li, Ran Meng, Tao Wang, Carole Dulong: Performance Analysis of Java Concurrent Programming: A Case Study of Video Mining System (2006)

[5]. Andreas Brunnert, Christian Vogele, Helmut Krcmar: Automatic Performance Model Generation for Java Enterprise Edition (EE) Applications (2013)

[6]. Sourabh Shastri: Studies on the Comparative Analysis and Performance Prediction of Sorting Algorithms in Data Structures (2014)

[7]. Sentry. [Online]. Available from: https://docs.sentry.io/platforms/java/[Accessed 20th August 2021]

[8]. JMeter. [Online]. Available from: https://jmeter.apache.org/[Accessed 20th August 2021]

[9]. Java VM. [Online]. Available from https://docs.oracle.com/javase/specs/jvms/se7/html/[Accessed 20th August 2021]

[10]. Visual VM. [Online]. Available from: https://visualvm.github.io/documentation.html[Accessed 20th August 2021]

[11]. Javasisst. [Online]. Available from: https://www.baeldung.com/javassist[Accessed 2nd September 2021]

[12]. Marten Vooberg: A performance analysis for membership data structures for integers in Java (2021)

[13]. Matúš Sulír, Michaela Baˇcíková, Matej Madeja, Sergej Chodarev and Ján Juhár: Large-Scale Dataset of Local Java Software Build Results(2020)

[14]. Classmexer. [Online]. Available from: https://www.javamex.com/classmexer[Accessed 5th March 2022]

[15]. Java Instrumentation. [Online]. Available from: https://www.baeldung.com/java-size-of-object[Accessed 5th March 2022]

[16]. Github Crawler for Java Programs. [Online]. Available from: https://github.com/janithahn/github_crawler.git[Accessed 7th April 2022]

[17]. Basic Benchmark Implementation. [Online]. Available from: https://github.com/janithahn/Benchmark.git[Accessed 7th April 2022]

[18]. Benchmarking JAR File. [Online]. Available from: https://github.com/janithahn/benchmark-jar.git[Accessed 7th April 2022]

APPENDIX

APPENDIX A: Screenshots from the research work process.

NumberOfElements	DataStructure	insertAt()
100000	ArrayList	10.76
100000	ArrayList_Initialized	5.23
100000	LinkedList	1.57
100000	Vector	10.81
100000	Vector_Initialized	5.18

NumberOfElements	DataStructure	insert()
100000	ArrayList	6.75
100000	ArrayList_Initialized	2.94
100000	LinkedList	8.39
100000	Vector	7.39
100000	Vector_Initialized	2.96

NumberOfElements	DataStructure	insert()	ds_insert()
100000	ArrayList	6.75	ArrayList_Initialized
100000	ArrayList_Initialized	2.94	ArrayList_Initialized
100000	LinkedList	8.39	ArrayList_Initialized
100000	Vector	7.39	ArrayList_Initialized
100000	Vector_Initialized	2.96	ArrayList_Initialized
...	...	...	...
10050000	ArrayList	17.17	LinkedList
10050000	ArrayList_Initialized	14.21	LinkedList
10050000	LinkedList	6.86	LinkedList
10050000	Vector	10.05	LinkedList
10050000	Vector_Initialized	8.65	LinkedList

```
Accuracy score:
 0.803030303030303
Confusion matrix:
 [[  5   2   4   0   1]
 [  1  91  15   1   4]
 [  0  14 139   1   1]
 [  0   9   2   7   2]
 [  0   4   3   1  23]]
Classification report:
                       precision    recall  f1-score   support

           ArrayList     0.8333    0.4167    0.5556        12
ArrayList_Initialized     0.7583    0.8125    0.7845       112
          LinkedList     0.8528    0.8968    0.8742       155
              Vector     0.7000    0.3500    0.4667        20
   Vector_Initialized     0.7419    0.7419    0.7419        31

            accuracy                         0.8030       330
           macro avg     0.7773    0.6436    0.6846       330
        weighted avg     0.8003    0.8030    0.7950       330

Precision:
 [0.83333333 0.75833333 0.85276074 0.7        0.74193548]
Recall:
 [0.41666667 0.8125     0.89677419 0.35       0.74193548]
```

```
Accuracy score:
 0.8620689655172413
Confusion matrix:
 [[25  0  0  1]
 [ 2 41  1  2]
 [ 0  0 17  1]
 [ 3  6  0 17]]
Classification report:
                          precision    recall  f1-score   support

                 HashMap     0.8333    0.9615    0.8929        26
     HashMap_Initialized     0.8723    0.8913    0.8817        46
           LinkedHashMap     0.9444    0.9444    0.9444        18
LinkedHashMap_Initialized     0.8095    0.6538    0.7234        26

                accuracy                         0.8621       116
               macro avg     0.8649    0.8628    0.8606       116
            weighted avg     0.8607    0.8621    0.8585       116

Precision:
 [0.83333333 0.87234043 0.94444444 0.80952381]
Recall:
 [0.96153846 0.89130435 0.94444444 0.65384615]
```