Czech Technical University

Faculty of Electrical Engineering

Department of Computer science

Bachelor's thesis 2015/2016

Java Collections Performance

Petr Tománek

Supervisor: ***Ing. Petr Aubrecht, Ph.D.***

June 2015/2016

Acknoledgement

I would like to thank my family, my girlfriend and my friends for support. They have supported me for the whole time and I'm really greatful for that.

I want to give special thanks to Ing. Petr Aubrecht, Ph.D. for patience, the knowledge and for leadershiping the project. It was an honor to work with you.

Declaration

Hereby I declare that I created this bachelor thesis independently under the leadership of Ing. Petr Aubrecht, Ph.D. and that I stated every literature and other source that I used. I agree with the publication of this bachelor thesis.

In Prague on day ....

Contents

1 Introduction **1**
1.1 Targets of Thesis . . . 1
1.2 Future in Information Technology . . . 2
1.3 Personal Motivation . . . 2

2 Analysis of Problem Domain in Broader Sense **3**
2.1 Data Storages . . . 3
2.2 Data Types . . . 3
2.3 Comparability . . . 4
2.4 Abstract Data Structures . . . 6
2.5 Efficiency of Algorithms . . . 8

3 Analysis of Problem Domain in Narrower Sense **13**
3.1 Implementations of Data Structures . . . 13
3.2 Types of Sorting . . . 19
3.3 Parallel Collections . . . 21

4 Analysis of Technological Possibilities **23**
4.1 Collections Frameworks . . . 23
4.2 Utility Classes . . . 26
4.3 Distributed Caches . . . 28

5 Analysis and Assesment of Test Cases **29**
5.1 Microbenchmarking Framework . . . 29
5.2 Analysis of Test Cases . . . 29

6 Design of Test Cases **31**
6.1 Division of Test Cases into Operations . . . 31

7 Java Collections Framework – Basic Collections **33**
7.1 Java Collections Framework – Interfaces and Operations . . . 33
7.2 Java Collections Framework – List Interface Performance . . . 33
7.3 Java Collections Framework – Set Interface Performance . . . 47
7.4 Java Collections Framework – Map Interface Performance . . . 52
7.5 Java Collections Framework – Results anomaly . . . 60
7.6 Java Collections Framework – Parallel Collections . . . 61

8 Java External Collections Frameworks **62**
8.1 List Interface Variants Performance . . . 62

9 Conclusion **66**
9.1 Personal Influence . . . 66
9.2 Follow-up Work . . . 66

10 Bibliography **67**

Abstrakt

Hlavní cíl této bakalářské práce je vysvětlení a porovnání datových struktur dostupných pro běžného programátora v Javě. Soustředil jsem se na kolekce, které jsou poskytovány v Java Virtual Machine a také na některé často používané open source frameworky.

Všechny testy jsou zobrazeny v grafech podle výkonu při vstupní velikosti dat. Je jednoduché porovnat různé implementace, vliv vylepšení v Java 8, atp.

Tato teze by měla objasnit mýty kolem kolekcí v Javě. Například mnoho lidí používá nevhodné struktury jako LinkedList na místo ArrayListu, protože neznají opravdovou efektivitu zvětšování pole v ArrayListu.

Tato práce by měla být základem pro budoucí testování paralelního zpracování dat, možná i frameworků zabývajících se distribuovaným zpracováním.

Klíčová slova

Kolekce, **Porovnání**, Efektivita

Abstract

The main aim of this bachelor thesis is explanation and comparison of structures for data storage which are available for regular Java programmer. I have concentrated on collections that are provided Java Virtual Machine and also several frequently used open source frameworks.

All the tests are shown in graphs per performance for input size. It is easy to compare various implementations, influence of improvements in Java 8, etc.

This thesis should clarify the myths around Java collections. For example many people use inappropriate structures like LinkedList instead of ArrayList, because they do not know about real efficiency of enlarging array in ArrayList.

This work should be base for the future investigation of tests in parallel data processing, maybe distributed frameworks.

Keywords

Collections, **Comparing**, Efficiency

1 Introduction

This bachelor thesis aims to deliver foundation for usage of structures in Java programming language – specifically collections for data storage. In many cases it is hard to use something that was built before because you do not understand how it works. It is even harder to understand how it was done and how it works. Although as soon as programmer understands the principle, how it was done and how it works, he can use it with greater quality. The knowledge of "how" can give him much more than just usefulness of itself.

"Scientia potentia est" also known as *"Knowledge is Power"*.
– Sir Francis Bacon

In my area of work experience, many people consider themselves as programmers but without knowledge they either make poor choices, work slowly or just make a code less readable and not as effective.

To make right decision, or close to, it is necessary to know where are strengths of designs, implementations and what can you expect in extreme cases. To understand principles it requires hard work. Reading huge documentations and understanding the whole subject can be lengthy. The best case is to be shown how to with brief documentation. Of course with good designs it is evident what you can do and how it works, so documentation is not key, but in many cases the problem is not trivial and so it is not as evident from design and documentation is absolute necessity to understand the principle.

1.1 Targets of Thesis

In this bachelor thesis I aim on multiple targets. First target is to describe basic data structures implemented in Java Collections Framework and measure how well they are implemented. My second target is to measure quality of data structures that work with multiple threads from Java Collections Framework. My third target is to compare and measure various collections that extend, enhance or add the additional functions to Java Collections Framework. Fourth target is to briefly look at the caching and describe basic idea behind it.

This bachelor thesis is divided into chapters according to the targets. In analysis I am going to concentrate on describing data types and basic structures in Java. I will introduce collections technologies, libraries and frameworks available in Java either in official Java Development Kit or via importing as additional projects, eg. from Maven.

In design I will describe what operations are available and what operations are going to be tested in case study and how. I will describe what framework I'm going to use and how it is working. Next there will be description how to use this case study to test collections on your computer.

Implementation of case study, which is testing and benchmarking application, is divided into multiple parts. Collections and cache implementation are main parts with division of collection into sub-parts. Collections sub-parts are going to be Java Collections Framework basic implementations, Java Collections Framework multi-threaded, parallel, implementations and described various collections frameworks not included in Java Development Kit.

The Cache implementation is going to be simple usage of the Caching frameworks with mentions of possible extensions for future study of the subject.

1.2 Future in Information Technology

In past Central Processing Units had only one core and so usage of collections was simply about serial processing of elements and handling any operation was one at a time. With few cores it was not as huge problem to use just one thread to carry all the operations on collections. But further development of Central Processing Units is introducing more and more cores.

Nowadays (2016) it is good to use multiple cores to work with collections, eg. rendering multiple objects or counting equations. With ascending number of cores grows the importance of parallel usage of algorithms and structures. It will be important in the future to use collections wisely not just time and memory efficiently but thread-safely as well.

1.3 Personal Motivation

At my early stages of working experience I worked in website development. I considered frameworks as something that I should avoid. It was because of the reason that they were aiming for one too many things and almost none of them were created well enough to be used. As my experience grown I realized that creating every piece of code by myself was wasting time, because most of the code was already written and reuse would be much better. Later in my career I started to dig deeper into Java and I discovered that I should not banish all the frameworks because of my early experience and I started to work with some that are described in this thesis.

I do not have as huge spectrum of experience and this bachelor thesis was one of the best ideas to improve my skill set and to give me better understanding of what I have been using for some time.

Although I think optimization is important and most of the time is overlooked I tend to try to optimize too much. Some people call it "premature optimization" and I have to agree.

"We should forget about small efficiencies, say about 97% of the time: ***premature optimization is the root of all evil.*** *Yet we should not pass up our opportunities in that critical 3%"*
– Donald Knuth[1]

The purpose of this bachelor thesis is to improve quality of software (written in Java programming language) by giving more information about structures. Therefore optimize applications with better memory, time and thread-usage efficiency, not just to be thread-safe. As well to prevent premature optimization on collections, because they are one of the essential parts of any Java application.

[1] Knuth, Donald (December 1974). "Structured Programming with go to Statements"

2 Analysis of Problem Domain in Broader Sense

The problem domain in this thesis is a storage of data, containing data in data structures and the functions of data structures and efficiency.

2.1 Data Storages

In Information Technology, more specifically in Computer Science there are multiple variations of architectures for an electronic digital computer. As one of the first architectures there was a Von Neumann architecture. This architecture had stored instructions and data in memory unit, it was possible to fetch instruction or operate with data only one at a time. The Harvard architecture has improved over Von Neumann in the way that it can fetch instructions and work with data both at a time.

Modern computers can be seen as both von Neumann and Modified Harvard architecture. They use separate instruction and data caches, which is classified as Modified Harvard architecture although they use a unified address space, which is classified as von Neumann architecture. Although terminology of architectures is big concern some specialists qualify modern processors as 'Almost-von-Neumann' architectures and some as 'Modified Harvard'.[1]

Saved data can be accessed from processor's cache, although in processor cache it is possible to save just small amount of data. In processors there are L1, L2 and sometimes even L3, also known as multi-level, caches. In processors cache are latest used, or most often used data and this way the computer can work quite fast. It is far more faster then saving or accessing data from RAM. The downside is that processor cache has much less memory space for data than RAM. Even bigger amount of data can be saved on secondary storage like hard drives or later introduced solid state disks. They work slower than RAM and therefore have lot of data saved higher in order.

The latest chapter for data storage is Distributed data storage, with special case like Cloud storage. It is multi-computer data storage over the network. The major advantage over the mentioned storages is that it is possible to share data with multiple machines in almost any time. The down side is that for network transfer of data there is network overhead. Network overhead is additional information like headers, sequence numbers of the packets and some additional flags to ensure that data were received correctly. The obvious down side is that if there is no network it is not possible to access data. Another problem is that streaming data via network can have many steps from origin destination to the final destination which can be really slow. Further more there are some more problems like multiple machine accessing the same data at the same time or reading the data that are being changed at the time.

2.2 Data Types

Generally in programming there are a lots of **primitive data types**. They are called primitive or basic, because they are built-in in programming language. In Java Programming Language there are primitive data types that consist of numbers, letters and logical data type. The numerical primitive data types are either decimal or integer numbers. The integer numbers are short, integer and long, the decimal are float and double. The char, which is data type describing letter, is consisting of just one letter, number or sign. Last primitive data type is logical data type. It is called in Java programming language boolean. It has just two values true-false. Some languages denote it as 0 for false and 1 or anything else for true.

[1] https://en.wikipedia.org/wiki/Harvard_architecture#Contrast_with_modified_Harvard_architecture

Listing 2.1: Primitive Data Type int

```
int number;
number = 1;
```

This can not be be said about the objects. While primitive data types are stored as actual values whereas Object class is **Reference data type** which stores address of the object it refers to. Objects have a state, in which its primitive data types and other objects are, eg. name, color or speed, and behaviour, in Java methods of object that provides additional functions, eg. change objects state like slowing down speed.

Listing 2.2: Reference Data Type Integer

```
Integer number;
number = 1;
```

From Java 1.5 it is used for auto-boxing (2.2). As reference data type can inherit parents structure by interface or abstract class, eg. Car is inheriting Vehicles structure (2.3), it can have many variables and methods to, implicitly or explicitly, use it's variables. This means the reference data type can contain multiple primitive or even the reference data types. This allows the use of Generics and as well usage of the data structures. Default value of Reference data type in Java is null.

Listing 2.3: Reference data type Car extending Reference data type Vehicle

```
Vehicle vehicle;
vehicle = new Car();
```

2.3 Comparability

Many structures use comparing methods to add elements or sort them. In Java there are two main comparing methods.

As primitive types have always value and objects depend on implementation I will omit null values for comparing in this chapter.

2.3.1 Comparability and Natural Ordering

The primitive data types are easy to be sorted, because it is easy to compare them. While primitive types are comparable by standard Java relational operators (less than, less than or equal, greater than and greater than or equal), the objects have standard interface **Comparable** and for additional types of comparing **Comparator**. Comparable interface only consists of *compareTo(T x)* method while Comparator interface has method *compare(T x, T y)*.

In contrast to the primitive types and auto-boxed primitive types there are objects that can consist of multiple primitive types, or even objects. In this case it is impossible to set universal comparing method automatically to compare any object to another.

Comparation of primitive types using standard relational operators and implementation of Comparable interface in objects are called **Natural Ordering**.

For example int auto-boxing class Integer implements *compareTo(Integer y)* method and it calls method *compare(int x, int y)* (although the method compare is not implementation of Comparator interface). From Java 1.7 the implementation looks like 2.4, where y is the number that is to be compared with variable x that is saved in Integer object.

Listing 2.4: Comparable interface implementation in Integer class

```
public int compareTo(Integer y) {
    return compare(this.num, int y)
}

private int compare(int x, int y) {
    return (x < y) ? -1 : ((x == y) ? 0 : 1);
}
```

The important thing is that method compareTo(T y) has to be designed by implementor to be as stated in its documentation [2]:

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure sgn(x.compareTo(y)) == –sgn(y.compareTo(x)) for all x and y. (This implies that x.compareTo(y) must throw an exception iff y.compareTo(x) throws an exception.)

The implementor must also ensure that the relation is transitive:
(x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.

Finally, the implementor must ensure that:
x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.

It is strongly recommended, but not strictly required that:
(x.compareTo(y)==0) == (x.equals(y)).

Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation sgn(expression) designates the mathematical signum function, which is defined to return one of –1, 0, or 1 according to whether the value of expression is negative, zero or positive.

The Comparator interface add functionality of having other type of ordering than Natural Ordering. Documentation states for *compare(int x, int y)* method similar description to the *compareTo(Integer y)* method documentation.

Comparator interface implementation can be added to tree structures as well as to the sort methods as parameter to set ordering or switch from the Natural Ordering.

2.3.2 Hash Function

Hash function is function mapping data of arbitrary size to fixed size representing the mapped data. Hashing has advantage because it is much faster to find item in collection using shorter hashed value then to find it using the original values. Hash function of objects is used very in multiple structures.

Because every object implementation has fundamental inheritance of Object class with pre-implemented hash function it is not necessary to implement the hash function although it is recommended to create hash function according to specific purpose of class, eg. make it cryptographically secure. The Object class has implementation of *hashCode()* and *equals(Object obj)*. Both methods are very important.

[2] https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

The method *hashCode()* documentation in Object implementations states [3]:

The general contract of hashCode is:

Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

The method *equals()* documentation in Object implementations states [4]:

The equals method implements an equivalence relation on non-null object references:

It is reflexive: for any non-null reference value x, x.equals(x) should return true.

It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.

It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.

It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

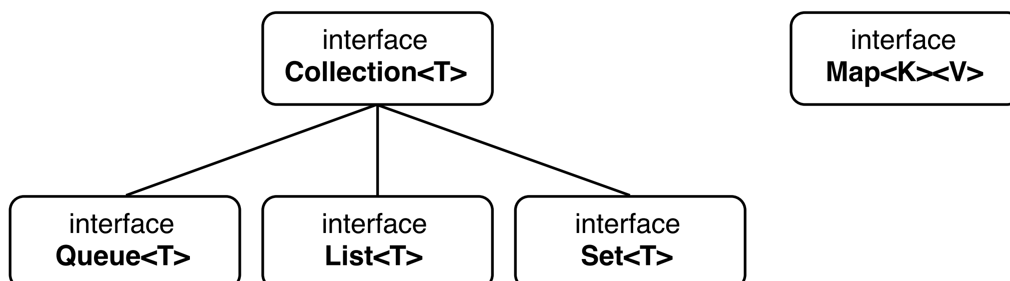
For any non-null reference value x, x.equals(null) should return false.

The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).

2.4 Abstract Data Structures

With Reference data types it is possible to create data structures. There are multiple data structure interfaces that can be implemented. In programming languages data structures are often called Collections or Containers prior to the longer name Data structures.

Figure 2.1: Collection interfaces in Java Collections Framework

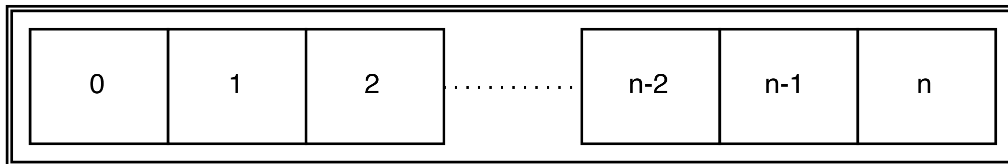


[3]https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html
[4]https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

2.4.1 Array

It is possible to use **Array** of objects. In many programming languages it is considered as primitive type, but Java implements array class for each array type in JVM. Arrays allow duplicate values of objects and generally they do not have to be ordered.

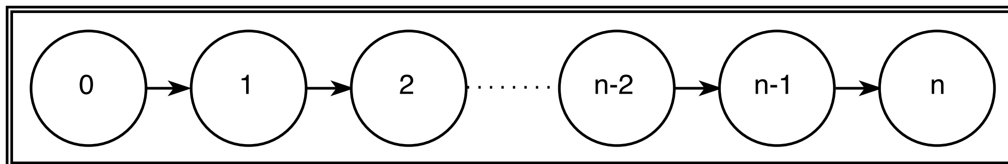
Figure 2.2: Array of objects



2.4.2 List

First data structure from Collections interface is **List**. It was introduced in Java 1.2 and is member of the Java Collections Framework. List is an object sequence. This means, that objects in list are linked in order depending on implementation. List can contain duplicates and it is possible to find element by position in the list.

Figure 2.3: List of objects



2.4.3 Set

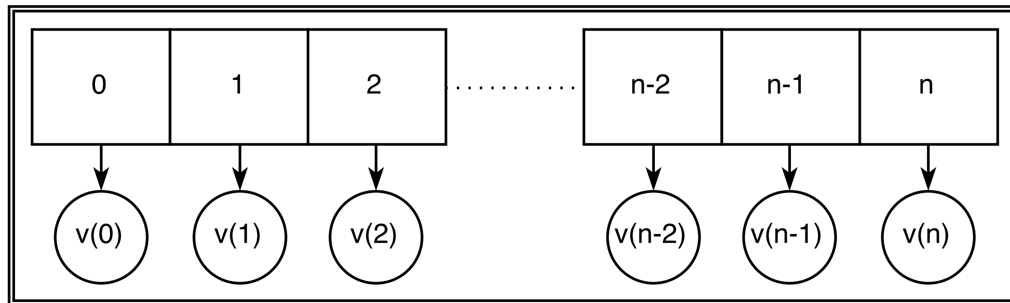
Next data structure from Collections interface is **Set**. It was introduced in Java 1.2 as member of Java Collections Framework. It is special in case of adding duplicate values, because this data structure disallows them. It is even possible, that adding the duplicate value will throw exception. In matter of ordering it depends on the implementation.

$$\forall(k_i, k_j \in S) : (i \neq j) \Rightarrow (x_i \neq x_j) \tag{2.1}$$

2.4.4 Map

Next abstract data structure from Collections interface is **Map**. It was introduced in Java 1.2 as member of Java Collections Framework. In contrast to the Set, which is containing only Keys, a Map contains Keys ans Values. To find the specific value Map has to disallow use of the duplicates in Keys as well as Set. It is not possible for basic map to map multiple values to one key.

Figure 2.4: Map of objects



2.4.5 Queue

Last data structure from Collections interface is **Queue**. It was introduced in Java 1.5 as member of Java Collections Framework. It is in some list implementations, but it has much more implementations of Queues, resp. Dequeues. Queue is another sequential data structure. It has added operations that instead of throwing exception when after failing return special values (either null or false). Those operations will be described later in this chapter. As an abstract data structure it is possible to list the Double ended queue also known as Dequeue.

Queue is similar to List interface except it adds additional functions that are null safe. The additional methods in this Collection interface are null safe. This means that instead of throwing exceptions after failing are returning different values. The methods are for insertion method offer(e), for remove method poll() and for examination method peek(). More detailed description of additional functions is in section **Queue implementation**.

Since Queue was added in JDK1.5 as subinterface of the Collection interface there were no structures added afterwards.

2.5 Efficiency of Algorithms

Using structures and big amount of data it is necessary to look at the efficiency, especially in most used parts. Every decision, like algorithm, that is repeated multiple times can increase time spent and memory consumption by huge amount. It is crucial to implement code almost flawlessly if it is repeated many times or with big data. Imagine there is one construction of new object that is not necessary and can be omitted. It will not matter for few usages, but as n for number of repeats can go to thousands and more it could slow down the process or fill the memory.

It is not simple to measure the complexity of algorithm. In fact it is not agreed how the complexity of algorithm should be described and measured. There are hundreds of ideas how complexity of algorithm should be described, some of them are commonly used and will be described in this chapter. There are some ways to do it accurately in theory and in practice and I will chose types of complexities that are the best possible in my opinion. The chosen complexities are commonly used and are theoretical complexity described by asymptotic orders of growth, theoretical hit-count of the values in memory, practical measuring of time for operations and practical hit-count on values in memory for operations.

2.5.1 Theoretical Complexity

It is possible to count the theoretical complexity of algorithm but in practice with algorithms implemented it may and often will differ. To count the theoretical complexity of algorithm described by asymptotic orders of growth it is necessary to count number of calls in one operation in algorithm as well as n-operations where $n > 1$ and estimate the asymptotic growth of an algorithm using limit of a function.

As described theoretical complexity of algorithm is not the same as practice complexity of algorithm it is absolutely crucial to know the theoretical complexity in order to predict the practical complexity. It is then possible, after determining both theoretical asymptotic orders of growth as well as hit-count of the values in memory of multiple algorithms, to compare the values of the theoretical complexities and therefore suggest the quality of the algorithms in general as well as in quality of algorithm under special circumstances. The information acquired in theoretical complexity adds the opportunity to choose the best algorithm to suit the given parameters. Theoretical complexity can give a hint how fast should the algorithm be in practice.

2.5.2 Big-O Notation

The Big-O notation is a function describing asymptotic behaviour of the algorithm. This provides an idea of how fast the algorithms function grows. It is used in multiple areas of work like mathematics or complexity theory. [1]

Function $f(x)$ is in asymptotic order of function $g(x)$ only if there is a positive constant c and real number x_0 that $|f(x)| \leq c|g(x)|$ for all $x \geq x_0$.

While omiting lower asymptotic growth rates when greater are present this function easily creates upper limit of function. Big-O notation describes the worst case scenario of algorithm. In following Figure 2.5 there are shown some of the asymptotic growth rates that are commonly encountered.

Figure 2.5: Big-O notation

2.5.3 Amortized Complexity

In this bachelor thesis, I will use term *amortized complexity*. The motivation for amortized complexity is not to look at the worst-case scenario, which can be too pessimistic. Instead the worst-case scenario, which happens once in many executions of operation.

The best description is as cited [2]:

In an amortized analysis, the time required to perform a sequence of data-structure operations is averaged over all the operations performed... Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the average performance of each operation in the worst case.

This complexity occurs for example in dynamic resizing of arrays like implementation of List interface in ArrayList.

2.5.4 Practice Complexity

In the practical complexity are factors that can change its complexity by multiples of the theoretical counted values. One of the main factors, maybe the main factor, for practical complexity is the implementation, although conversely even the best implementation can not help the theoretically bad algorithm to excel over those more efficient algorithms.

On the other side if the practical complexity is measured, for example practical measuring of time for operations by counting time for one and multiple operations, results can be different. The values from practically measured complexity are dependent on implementation and as well are dependent on case (can be somewhere between best case and worst case for the chosen algorithm). Those results would create wrong impression and therefore it is important to measure the time multiple times and create the average value and then estimate those values.

Table 2.1: Comparation of Different algorithms on different computers (in ms)

n (list-size)	Computer 1 run-time	Computer 2 run-time
16	8	100,000
250	125	200,000
4000	2000	300,000
64,000	32,000	400,000
1,000,000	500,000	500,000
16,000,000	8,000,000	600,000

In the Table 2.1 there are shown two computers with two different algorithms solving one problem. Computer 1 is the state-of-the-art machine, the Computer 2 is average computer. As the time complexity of the algorithm on Computer 1 is $n/2$ and the time complexity on the Computer 2 is $log(n) * 25000$. Via observation it is possible to see that the run-times for both computers the point where they are equal is 1,000,000. Where less then 1,000,000 objects the algorithm in Computer 1 is better, whereas over 1,000,000 the algorithm in Computer 2 becomes better. After observation it is easy to count theoretical complexities and choose the orders of growth. It is apparent that Computer 1 runs the algorithm with time complexity of n whereas Computer 2 runs the the algorithm with complexity $log(n)$ which is much better in terms of time consumption.

In general if one operation of algorithm runs for n seconds and second algorithm doing the same thing for m seconds, doing those cycles of algorithms p-times would increase their amount of time spent. In first algorithm the complexity could be in direct proportion to the amount of cycles therefore approximately $n \cdot p$. The second algorithm's complexity could be m^p. When

comparing those values and find out which one is faster for one cycle and for multiple cycles. We can try to do more cycles and compare those results and find approximate estimate the value.

2.5.5 Time Complexity

One of the biggest threats in efficiency is time consumption. As more and more applications use big data there comes greater emphasis of possible problems with retrieving and processing as it is about iterating over data. In this part every decision matters. Although time efficiency is one of the most important things in almost any application it is one of the easiest to be measured and therefore solved as every process can be measured. It is possible to measure the time complexity of code that handles data as well as measuring any algorithms time complexity. There are some matters that are more complex and has to be considered, eg. measuring the algorithm on state-of-the-art machine and on much slower machine will give absolutely different and inconsistent results. Another matter is that if there was done just one testing for each of multiple algorithms there could be some error, eg. during measurement some application can start working on machine unexpectedly like anti-virus or updating application and ruin consistency as well. One of the matters can be as was in Table 2.1 (on page 10) that with low number of elements or repeats one algorithm can be much better and therefore seem to be much more efficient, but with more and more data added the algorithm starts to worsen its efficiency. Therefore it is crucial to measure time complexity of multiple algorithms on the same machine multiple times with multiple different quantities To keep results consistent.

In Java programming language there are multiple ways how to measure time complexity, one that is used in this thesis is described in Section 5.1.

2.5.6 Storage Complexity

Very obvious threat is storage complexity which observes how well the application and more specifically algorithms handle the data. Although memory efficiency is not as visible it is as important as the time efficiency and maybe even more. There are several things that can turn out to be wrong. Problems in memory consumption can cost loss of time and even worse loss of information. In the best case there is visible usage of data, that can show what is happening in an application, but in most cases application gets frozen, throws an exception or in worst case Java throws an error. Not even can the high memory usage slow down the process, it can eventually overload the memory and Java would throw OutOfMemoryError. It is almost impossible to recover from such error without restarting the process and loosing all the applications context is almost inevitable.

Although it is not hard to calculate how well algorithm handles the data theoretically, eg. how algorithm handles saving, fetching or sorting the data, it is much harder to observe hit-count, or miss-count in practice. When choosing the algorithm to handle data it is important to be cautious and look for possibilities. For example in collections, as they were described in this chapter, in some cases queue is the best option, if what is wanted is iterating like in real queue and no sorting and searching is needed, other times the tree can prove to be the most efficient collection, with sorting and searching. In every case it is important to think about usage and try to predict what the algorithm should do in the future to chose the best possible implementation.

2.5.7 Parallel-use

Big problem, as it is kind of new and not observed enough is parallelism. In last two decades there was trend of increasing number of processor cores in computers and with it there came a precious possibility of multi threading. It gives an option for one thread to work on graphical user interface and others on using the data with some sort of permeation in result. It is great to have possibility of multiple threads processing applications features and data. It has a lot of pros but many cons that are not as easy to be handled. Examples of pros are evident with processing multiple algorithms at once the dominant point. On the other side examples of cons are that because of that each thread uses its own allocated part of the memory to not override the data of the other threads it is needed to have bigger memory, or sometimes the threads need to use the shared data for the whole application and either the data is serialized, that means it is set to be given in serial order and is not using the parallelism properly, or is given the data and there is a risk of the data being overwritten or having some parts added/deleted.

Although a lot similar problems have evolved in databases via parallel access of table rows it is not possible to solve parallelism in same fashion. The database problems were solved by locking the views (accessed tables or rows). It is possible to serialize the algorithms, but there has to be multiple locking steps. If the locking is not covered entirely there are some possible exceptions to be thrown like ConcurrentModificationException.

This part of complexity is not inspected well enough but as the numbers of cores are being increased it is starting to be an important issue to look at. This bachelor thesis should describe the collections in this particular sector and interactions in collections in respect of the usage of multiple cores.

3 Analysis of Problem Domain in Narrower Sense

More detailed view of the problem domain with implementation of data structures, sorting algorithms and description of parallel collections in JCF.

3.1 Implementations of Data Structures

As described in section Abstract Data Structures (Section 2.4, page 6) there are some major structure interfaces in Java. Every abstract structure has multiple implementations with different parameters and different time and memory complexity. In this section there will be description, time and memory complexity and basic interpretation of the preferred structure implementations. The structures described in this section are all implemented in Java Collections Framework and therefore they use its interface. Every Collection interface, except Map, inherits from interface, Listing 3.1 with Generic object E that specifies what object type will implementation of Collection contain.

Listing 3.1: Collection interface used in all implementations

```
public interface Collection<E> extends Iterable<E> { ... }
```

3.1.1 List

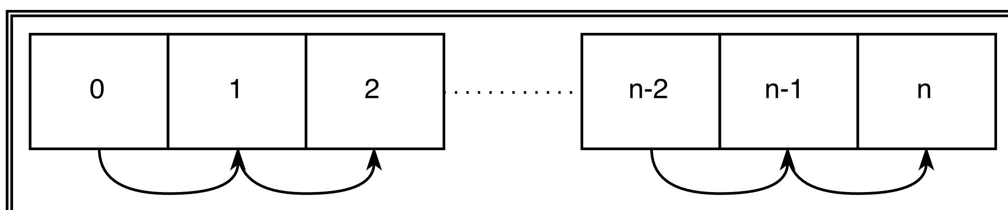
List structure is an ordered collection. The two preferred implementations of a List interface are ArrayList and LinkedList. There are some other implementation like Vector and its descendant Stack, but those two described implementations were added in JDK1.0 and are used very rarely so they will not be describe here.

3.1.1.a ArrayList

Documentation of ArrayList states [1]:

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides dynamicmethods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

Figure 3.1: Array list



Insert operation worst case performance – in most cases $\mathcal{O}(1)$ with occasional $\mathcal{O}(n)$ when array size increases; Amortized Complexity is $\mathcal{O}(1)$ (Section 2.5.3, page 10) due to way the array is enlarged – double the size.

[1] https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

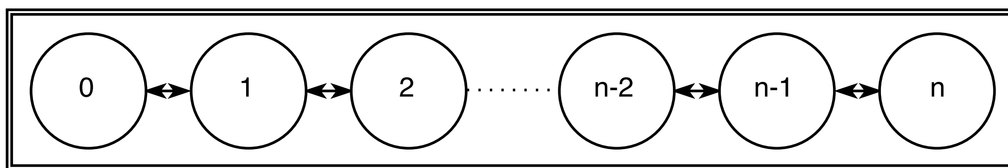
Get operation case performance – $\mathcal{O}(1)$
Remove operation best/worst case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Space complexity – $\mathcal{O}(n)$

3.1.1.b LinkedList

Documentation of LinkedList states [2]:

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

Figure 3.2: Linked list



All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Insert operation best/worst case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Get operation case performance – $\mathcal{O}(n)$
Remove operation best/worst case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Space complexity – $\mathcal{O}(n)$

3.1.2 Queue

Queue structure is ordered collection. Often used as FIFO, LIFO or sometimes priority Queue. Queues descendants vary from subinterfaces like dequeue to the PriorityQueue and LinkedList that was described in the List structure description. There are some thread-safe implementations like ConcurrentLinkedQueue and SynchronousQueue.

Additional methods that are null safe are for the insertion operation: offer(e) – returning boolean value, the removing operation: poll() – returning and removing the head of Queue of Queues generic type element E or null if Queue is empty and the examine operation: peek() – returning the head of Queue of Queues generic type element E without removing it or null if Queue is empty.

3.1.3 Set

Set is collection without duplicate elements. Usually not ordered. Most used are either without order HashSet, or the ordered EnumSet, LinkedHashSet and TreeSet. There are some thread-safe implementations like ConcurrentSkipListSet which are obviously ordered.

As some of the sets are heavily dependent on values of objects, they have big problem when objects inside the Set are being changed.

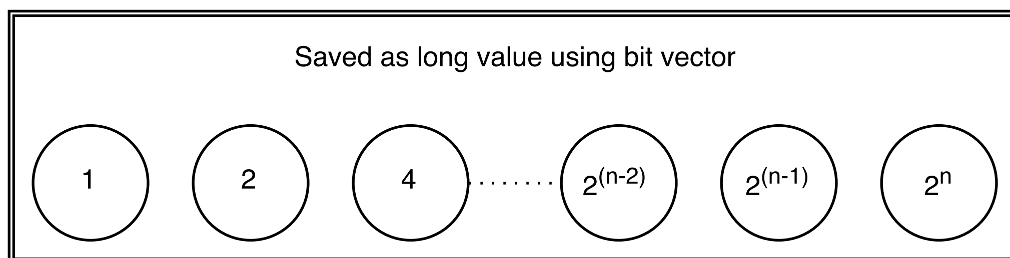
[2] https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

3.1.3.a EnumSet

Documentation of EnumSet states [3]:

A specialized Set implementation for use with enum types. All of the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created. Enum sets are represented internally as bit vectors. This representation is extremely compact and efficient. The space and time performance of this class should be good enough to allow its use as a high-quality, typesafe alternative to traditional int-based "bit flags." Even bulk operations (such as containsAll and retainAll) should run very quickly if their argument is also an enum set.

Figure 3.3: Enum set



Insert operation best/worst case performance – $\mathcal{O}(1)$
Get operation case performance – $\mathcal{O}(1)$
Remove operation case performance – $\mathcal{O}(1)$
Space complexity – $\mathcal{O}(1)$

3.1.3.b HashSet

Documentation of HashSet states [4]:

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

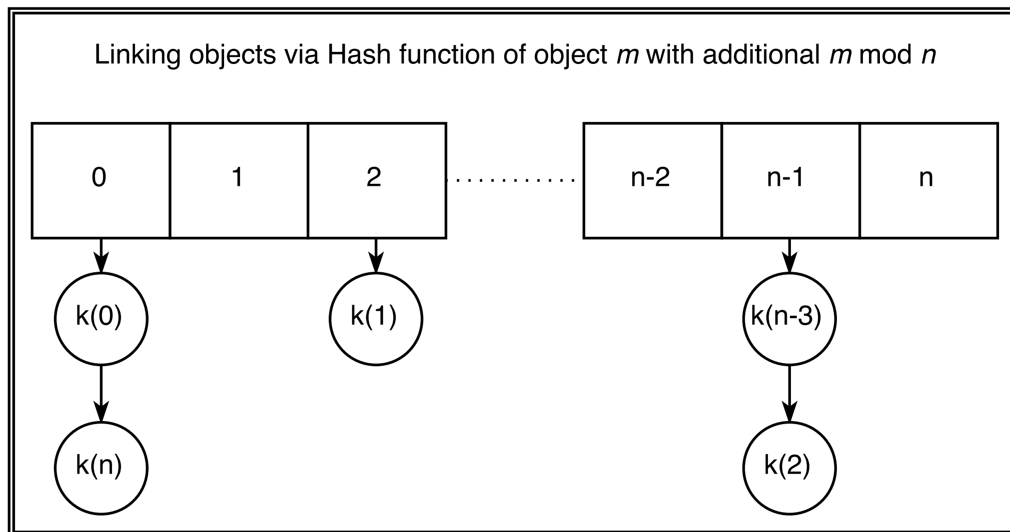
This Set uses hashing function to distribute elements in the hashing table. More about hashing in Hash Function (Section 2.3.2, page 5).

The case of function depends on quality of hash function. The worst cases can happen when the hash function is badly managed, eg. hash which returns constant. The best function will distribute all of the elements evenly in the hash table and so performance will be in constant time.

Insert operation best/worst case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Get operation case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Remove operation case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Space complexity – $\mathcal{O}(n)$

[3] https://docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html
[4] https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

Figure 3.4: Hash set



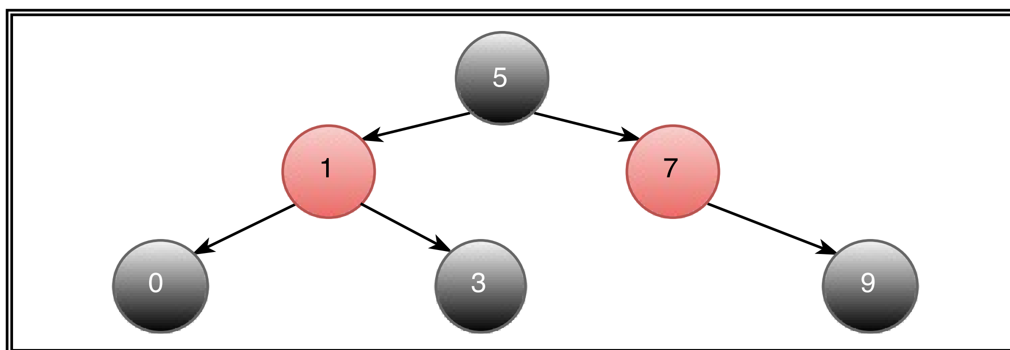
3.1.3.c TreeSet

Documentation of TreeSet states [5]:

A NavigableSet implementation based on a TreeMap. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

This Set uses comparing function to sort elements in the tree structure. More about comparing in Comparability and Natural Ordering (Section 2.3.1, page 4).

Figure 3.5: Tree set



TreeSets performance depends on chosen tree algorithm. The performances are for Red-Black Tree (Section 3.2.2.a, page 20) algorithm which is used as TreeSets implementation.

[5]https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html

Insert operation case performance – $\mathcal{O}(log(n))$
Get operation case performance – $\mathcal{O}(log(n))$
Remove operation case performance – $\mathcal{O}(log(n))$
Space complexity – $\mathcal{O}(n)$

3.1.4 Map

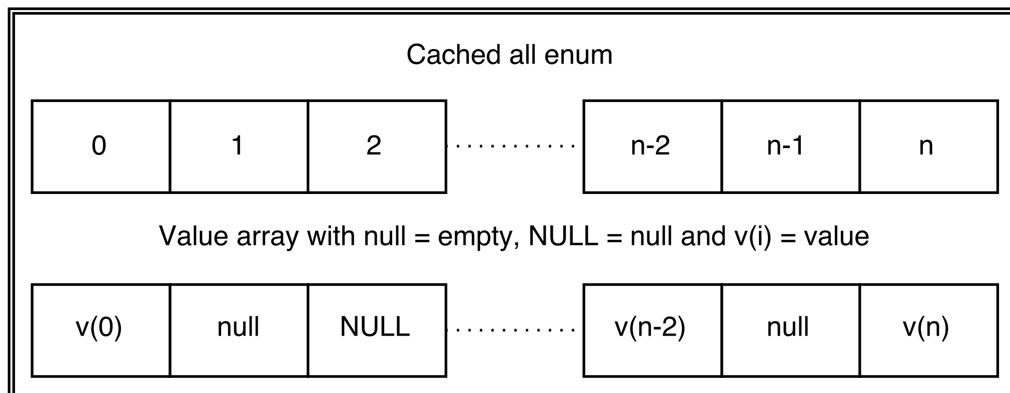
Map is collection structure holding key and value. Usually not ordered. Map is without duplicate keys. Most used are either unordered EnumMap and HashMap, or the ordered LinkedHashMap and TreeMap. Even the Map interface has some implementations that are thread-safe like ConcurrentSkipListMap and ConcurrentHashMap.

3.1.4.a EnumMap

Documentation of EnumMap states [6]:

A specialized Map implementation for use with enum type keys. All of the keys in an enum map must come from a single enum type that is specified, explicitly or implicitly, when the map is created. Enum maps are represented internally as arrays. This representation is extremely compact and efficient.

Figure 3.6: Enum map



Insert operation case performance – $\mathcal{O}(1)$
Get operation case performance – $\mathcal{O}(1)$
Remove operation case performance – $\mathcal{O}(1)$
Space complexity – $\mathcal{O}(n)$

3.1.4.b HashMap

Documentation of HashMap states [7]:

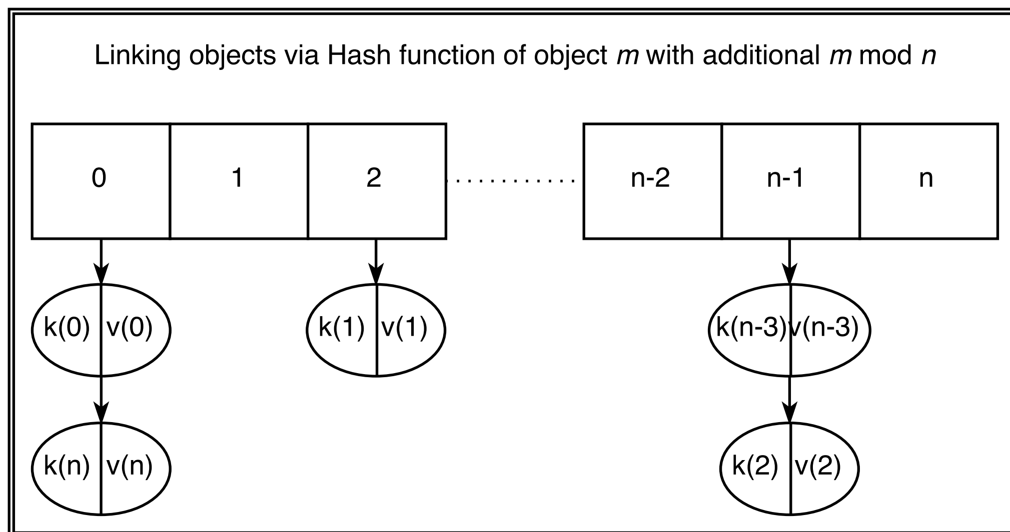
Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class

[6] https://docs.oracle.com/javase/8/docs/api/java/util/EnumMap.html
[7] https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This Map uses hashing function to distribute elements in the hashing table. More about hashing in Hash Function (Section 2.3.2, page 5).

Figure 3.7: Hash map



The case of function depends on quality of hash function. The worst cases can happen when the hash function is badly managed, eg. hash which returns constant. The best function will distribute all of the elements evenly in the hash table and so performance will be in constant time.

Insert operation best/worst case performance – $\mathcal{O}(1)/\mathcal{O}(n)$

Get operation case performance – $\mathcal{O}(1)/\mathcal{O}(n)$

Remove operation case performance – $\mathcal{O}(1)/\mathcal{O}(n)$

Space complexity – $\mathcal{O}(n)$

3.1.4.c TreeMap

Documentation of TreeMap states [8]:

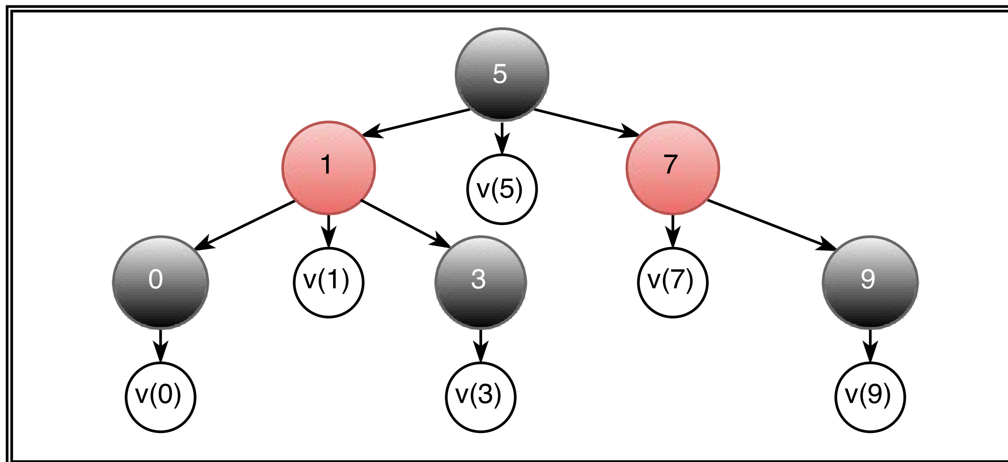
A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This Set uses comparing function to sort elements in the tree structure. More about comparing in Comparability and Natural Ordering (Section 2.3.1, page 4).

TreeMaps performance depends on chosen tree algorithm. The performances are for Red-Black tree which is the algorithm used for TreeMap implementation in JCF. More details in Red-Black Tree (Section 3.2.2.a, page 20).

[8] https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

Figure 3.8: Tree map



Insert operation case performance – $\mathcal{O}(log(n))$
Get operation case performance – $\mathcal{O}(log(n))$
Remove operation case performance – $\mathcal{O}(log(n))$
Space complexity – $\mathcal{O}(n)$

3.2 Types of Sorting

Sorting is operation that makes order using some keys that can be ordered in a specific way. To order a Collection there has to be some criteria by which the order can be kept. It is important, that the criteria for ordering are consistent at all times. If the consistency is not kept the order cannot be done, eg. if $3 > 1$ and at the same time $1 = 3$ would be really confusing. In Java it is possible to order using Comparator for the specified objects or some ordering implemented in Objects, most commonly natural ordering that is implemented in Comparable interface.

Sorting is a function of comparing values one to another and moving their order index up or down. There are many possible implementations of sorting a collection of values. Some Collections are ordered because they are using linking with Nodes so they keep order of adding, some are sorted in Tree structures and some are not ordered at all.

To investigate what sorting algorithm to chose in what situation it is important to know how efficient in time and storage it is.

3.2.1 Sort Algorithms

3.2.1.a Insertion sort

Insertion sort is stable sorting algorithm based on comparing sorted values. With asymptotic order of growth $\mathcal{O}(n^2)$ it is one of the worst, but if the collection to be sorted is almost sorted the order of growth comes close to $\mathcal{O}(n)$ with almost no moves and just walkthrough. This is the reason the Insertion sort is used with combination of other algorithms.

Worst/best case performance – $\mathcal{O}(n^2)/\mathcal{O}(n)$
Worst case space complexity – $\mathcal{O}(n)$ with additional $\mathcal{O}(1)$

3.2.1.b Merge sort

Merge sort is stable sorting algorithm of recursion type divide and conquer. This algorithm has order of growth equal to $\mathcal{O}(n\ log(n))$. With additional array of size n this sort first breaks down the collection to the smallest unit. Then sorts the unit with adjacent unit via sorting each of its values and then merges then to one bigger unit.

Worst/best case performance – $\mathcal{O}(n\ log(n))/\mathcal{O}(n\ log(n))$

Worst case space complexity – $\mathcal{O}(n)$ with additional $\mathcal{O}(n)$

3.2.1.c Quicksort

Quicksort is non-stable sorting algorithm of recursion divide and conquer. This algorithm has order of growth equal to $\mathcal{O}(n^2)$. Quicksort algorithm uses pivot-based comparison. After choosing pivot on one side are the smaller elements and on the other side are the bigger elements. Dividing the collection into the smaller units until the collection is sorted.

Worst/best case performance – $\mathcal{O}(n^2)/\mathcal{O}(n\ log(n))$

Worst case space complexity – $\mathcal{O}(n)$ with additional $\mathcal{O}(log(n))$

3.2.1.d Dual-Pivot Quicksort

Modified Quicksort which is not stable and uses the recursion divide and conquer. For smaller arrays it uses Insertion sort, for other arrays Dual-Pivot chooses two pivots where first pivot is smaller then the second pivot. All elements are compared until they are moved into according parts where part 1 is less then Pivot one, part 2 is between Pivot one and Pivot two and part 3 is bigger then Pivot two whereas part 4 consists of unsorted elements.

Worst/best case performance – $\mathcal{O}(n^2)/\mathcal{O}(n\ log(n))$

Worst case space complexity – $\mathcal{O}(n)$ with additional $\mathcal{O}(log(n))$

3.2.1.e Tim sort

Tim sort is stable sorting algorithm that combines Merge sort and Insertion sort. This algorithm has order of growth equal to Merge sorts $\mathcal{O}(n\ log(n))$ but with combination of Insertion sort it has best case equal to $\mathcal{O}(n)$ only if the collection is already sorted. Tim sort increases the performance of sorting by using Insertion sort where it is possible to find longer pre-sorted units.

Worst/best case performance – $\mathcal{O}(n\ log(n))/\mathcal{O}(n)$

Worst case space complexity – $\mathcal{O}(n)$

3.2.2 Tree Sorts

There is multiple existing tree structures that are presorted. In Java as in tree structures sorting is done by Red-Black Tree.

3.2.2.a Red-Black Tree

The Red-Black tree is self-balancing binary search tree. It is also known as RB-Tree. To provide approximate balance in the tree the Red-Black tree structure adds additional bit in every node for color (red or black). Except the colors the structre is almost identical to (uncolored) binary tree.

Insert operation case performance – $\mathcal{O}(log(n))$
Remove operation case performance – $\mathcal{O}(log(n))$
Get operation case performance – $\mathcal{O}(log(n))$
Worst case space complexity – $\mathcal{O}(n)$

3.3 Parallel Collections

The applications that use collections with multiple threads have to be programmed carefully. The basic Java collections described in Analysis of Problem Domain in Broader Sense (Chapter 2, page 3) have no thread-safe implementation therefore parallel usage is not safe. In all implementations that were described in Implementations of Data Structures (Section 3.1, page 13) the Java documentation states: *Note that this implementation is not synchronized.*

There are some options to be considered in case of parallel usage of collections. First option is synchronizing the basic collections with wrapper class which synchronises the regular collection. Second option is by using thread-safe collections.

3.3.1 Synchronized Collections

The description on Java documentation of named collections mention that if multiple threads access the collection and any of the threads would want to modify it the specified collection would have to be synchronized externally. The advice in Java documentation of the specified collections states that such collections should be encapsulated. If encapsulation manually is not possible the advice states that the collection could be "wrapped" using JCF utility class *Collections*.

Listing 3.2: Synchronization as adviced in documentation of TreeSet

```
SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));
```

These methods return synchronized, therefore thread-safe, collection. Although as stated in method *In order to guarantee serial access, it is critical that **all** access to the backing collection is accomplished through the returned collection.* and that it is imperative to manually synchronize collection when iterating over it.

Listing 3.3: Synchronization as adviced in documentation of synchronizedSortedSet method

```
SortedSet s = Collections.synchronizedSortedSet(new TreeSet());
    ...
synchronized (s) {
    Iterator i = s.iterator(); // Must be in the synchronized block
    while (i.hasNext())
        foo(i.next());
}
```

3.3.2 Concurrent Collections

The second option for JCF is in package java.util.concurrent which adds multiple concurrent collection implementations of JCFs interfaces. *Collections are so frequently used that various concurrent friendly interfaces and implementations of collections are included in the APIs.*[9]

[9] https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

The java.util.concurrent package adds new interfaces extending Collection, respectively Queue and Map, as well as many implementations. *These types go beyond the synchronization wrappers ... to provide features that are frequently needed in concurrent programming.*[9]

JCF implementations in package java.util.concurrent help avoid memory consistency errors, which means that data are inconsistent in between multiple threads eg. first thread changes state but second reads at the same time the old state, by defining happens-before relationship.

4 Analysis of Technological Possibilities

In this chapter will be description of Collections, Utility classes and Caches. Each of named sections will have description, capabilities and additional functions.

4.1 Collections Frameworks

Collection interfaces were described briefly in Abstract Data Structures (Section 2.4, page 6) and later in Implementations of Data Structures (Section 3.1, page 13) more detailed description of preferred implementations. Every collection library will have description of it's collection type interfaces, collection type implementations and it's additional functions and operations.

Additional Structure Interfaces

Many structures are implementing similar interface (with just a few differences) but under different naming convention.

One structure interface is called in many variations **Multiset**, others call the structure **Bag**. It is Set which is counting how many times key was added into the structure.

Name of the structure interface called **Multimap** is used in all variations that contain it. This structure contains one key and many values, often distributed in list.

Other structure interface is called sometimes **BiMap**, other times it is called **BiDiMap**. As name suggest it is bi-directional association map, where key is mapped to value and value is mapped to key.

Last structure interface is named **Table**. This structure addresses the need of using two keys with one value association.

4.1.1 Java Collections Framework in Java 6 and Earlier

Before Java 1.2 there were just a few usable data structure libraries. Structures that were used were array-s, Vector-s and Hashtable-s. To solve this missing part of not having any official Collections library there ejected a lots of Collection frameworks. Some tried to keep consistency with C++ Standard Template Library, others tried to introduce new functionality and consistency was not the key feature.

Problem was addressed in Java 1.2 by creating official library called Java Collections Framework, also known as JCF. JCF had three main interfaces – List, Set and Map with multiple implementations. JCF has become official reusable collections data structure library. In Java 1.5 has emerged new collection interface – Queue with some additional implementations.

All implementations described in Implementations of Data Structures (Section 3.1, page 13) are already implemented in Java 1.6.

4.1.2 Java Collections Framework in Java 7

Javas advancements in Collections which mostly changed how Java looks now were introduced in JDK 7 launched on July 6, 2011.

Changes were made to Java concurrency, hashing and in utility class sorting and addition of diamond operator.

Improvements over the older Java updates were in Concurrency in JSR 166. One of the important advancements in Java 7 was addition of new interface TransferQueue under

java.util.concurrent. It is refinement of BlockingQueue interface. With interface was added one implementation – LinkedTransferQueue.

In update 6 there were changes of Hashing function, respectively added alternative hashing function for Strings in maps with larger capacity (recommended value of threshold was 512). This change has affected implementations of maps and map-derived collection implementations (HashMaps, Hashtables and HashSets and their descendants).

The alternative hash function improves the performance of these map implementations when a large number of key hash collisions are encountered.[1]

4.1.3 Java Collections Framework in Java 8

New advancements in Java were introduced on March 18, 2014.

The major changes were applied in hashing, newly there were added Lamba Expressions with addition of streams.

Hashing changes from JDK 7 were reversed. Alternative hashing and parameter threshold were deleted. Instead to improve performance of large hash collections (HashSet and HashMap and their descendants), these collections use balanced tree instead of LinkedList-s as nodes [2]. This change affects only collection with elements that implement Comparable interface.

4.1.4 Apache Commons for Java

Apache Commons is huge framework consisting of multiple libraries. One of the libraries is Commons Collections which is based on Java Collections Framework. Commons Collections extends and sometimes augments Java Collections Framework.

Commons-Collections seek to build upon the JDK classes by providing new interfaces, implementations and utilities.[3]

Additional interfaces are Bag, BidiMap, MultiSet and MultiValuedMap. Additional implementations of collections are ordered maps and sets, composite collections and reference maps.

4.1.5 Trove for Java

Trove is *high performance collections* library that provides implementation of collections for primitive types, hashing function for maps and sets and iterators for primitive collections.

As cited from the official site[45]:

Provide "free" (as in "free speech" and "free beer"), fast, lightweight implementations of the java.util Collections API. These implementations are designed to be pluggable replacements for their JDK equivalents.

Whenever possible, provide the same collections support for primitive types. This gap in the JDK is often addressed by using the "wrapper" classes (java.lang.Integer, java.lang.Float, etc.) with Object-based collections. For most applications, however, collections which store primitives directly will require less space and yield significant performance gains.

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/collections/changes7.html
[2] http://openjdk.java.net/jeps/180
[3] https://commons.apache.org/proper/commons-collections/
[4] http://trove4j.sourceforge.net/html/overview.html
[5] https://bitbucket.org/trove4j/trove

4.1.6 High Performance Primitive Collections for Java

High Performance Primitive Collections for Java also known as HPPC is implementation of collections for primitive types and their derivatives which use fully or combination of primitive types and objects.

As cited from the official site[6].

While HPPC is not strictly modeled after Java Collections Framework (JCF), we did try to make the APIs look similar enough for comfortable use.

HPPC provides template-generated implementations of typical collections, such as lists, sets and maps, for all Java primitive types. The primary driving force behind HPPC is optimization for highest performance and memory efficiency.

Differences between JCF and HPPC are shown in these tables HPPC compared with JCF for primitive types (Table 4.1, page 25) and HPPC compared with JCF for reference types (Table 4.2, page 25). The tables are cited from a official site overview [7]:

Table 4.1: HPPC compared with JCF for primitive types

	Java Collections	**HPPC (primitives)**
bit sets	java.util.BitSet	BitSet
array-backed lists	java.util.ArrayList java.util.Vector	[type]ArrayList
stacks	java.util.Stack	[type]Stack
deques	java.util.ArrayDeque	[type]ArrayDeque
hash maps (dictionaries)	java.util.HashMap	[keyType][valueType]OpenHashMap

While HPPC is mainly about primitives, we also distribute the generic-based collections, so that the benefits of direct data store access are also available for collections of non-primitives.

Table 4.2: HPPC compared with JCF for reference types

	Java Collections	**HPPC (generics)**
bit sets	java.util.BitSet	n/a
array-backed lists	java.util.ArrayList java.util.Vector	ObjectArrayList<T>
stacks	java.util.Stack	ObjectStack<T>
deques	java.util.ArrayDeque	ObjectArrayDeque<T>
hash maps (dictionaries)	java.util.HashMap	ObjectObjectOpenHashMap<K><V> [keyType]ObjectOpenHashMap<V> Object[valueType]OpenHashMap<K>

4.1.7 Google Guava for Java

The project named Guava contains Google Core Libraries for Java. Guava is a successor to the older library Google Collections Library which was extension of JCF. It requires JDK 1.6 or

[6]https://labs.carrotsearch.com/hppc.html
[7]http://labs.carrotsearch.com/download/hppc/0.2.0-dev/api/overview-summary.html

higher and provides various libraries of Google's core libraries like collections, caching, primitive support and concurrency as well as others.

Guava extends JCF with its own collections and adds corresponding utility classes. Guavas utility classes will be described in section Utility class.

Guava collection provides various additions to JCF like immutable collections and some interfaces on top of JCF. Guava has its own specific interfaces. New collection interfaces are **Multiset**, **Multimap**, **BiMap** and **Table**.

4.1.8 Standard Template Library in C++

Standard Template Library is collections library for C++ and is also known as STL. STL provides mainly collections (in C++ its called containers), iterators and utilities (called algorithms). Many of the collection types are fairly similar to JCF collections with exception in naming conventions of the collections.

4.2 Utility Classes

One of the parts in collections are utility classes. Utility classes are adding functionality which is not supported by basic collection implementations. One of the reasons for separation is that the methods are same for whole collections type or even collections and so they can be reused without reimplementing them in every collection implementation.

4.2.1 Java Collections Framework

In JCF the corresponding Utility classes are **Arrays** and **Collections**. One of many functions in these classes is sorting.

Documentation of Arrays in official site [8]

This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.

Documentation of Collections in official site [9]

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

The Java Arrays.sort() uses Dual Pivot Quicksort. Collections.sort() implementation changed (will be described later in this section).

4.2.2 Java Collections Framework in Java 6 and Earlier

The initial implementation of Collections.sort() used Merge sort.

4.2.3 Java Collections Framework in Java 7

Newly added were emptyIterator() methods and emptyEnumeration().

One of the important improvements was that diamond operator was added. This affected constructors because instantiating generic classes can be less verbose instead of older explicit specification.

Collections.sort() was changed from Merge sort to Timsort.

[8] `http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html`
[9] `http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html`

4.2.4 Java Collections Framework in Java 8

Java 8 added Lambda expressions and Streams. This allows functional programming in Java with possibility of parallel processing. There were also many minor additions in Collections.[10]

Tests of these features is in the plan for future.

4.2.5 Apache Commons for Java

Apache Commons library has multiple Utility classes. For every collection type interface one and many more. The utility classes provide utility methods and decorators to corresponding collections with CollectionUtils for Collection interfaces, ListUtils to List interfaces, QueueUtils to Queue interfaces, SetUtils to Set interfaces and MapUtils to Map interfaces.

One of the additions in utilities in Apache Commons is MapIterator interface and its implementations.

Additionally Apache Commons library has package functors which adds Closure, which is executed from inside of block, function or iteration, Predicate, which is object equivalent to if statement, Transformer, which converts input object to the output object without changing the input object, and Factory, which creates object without input parameter, interfaces and their implementations. Standard implementations of these utility interface are provided by their utility class. For Closure interface it is ClosureUtils, for Predicate interface it is PredicateUtils, for Transformer interface there is TransformerUtils and for Factory interface there exists FactoryUtils.

4.2.6 Google Guava for Java

Google Guava creates wrapper class for every primitive data type. Guava implementation of wrapper classes are utility classes that enhance Javas original wrapper classes, eg. for int Ints, UnsignedInteger or UnsignedInts.

They are well described on official site [11]:

These types cannot be used as objects or as type parameters to generic types, which means that many general-purpose utilities cannot be applied to them. Guava provides a number of these general-purpose utilities, ways of interfacing between primitive arrays and collection APIs, conversion from types to byte array representations, and support for unsigned behaviors on certain types.

Guava extends JCF with its own collections and adds corresponding utility classes. List has in Guava utility class Lists, Set and SortedSet has utility class Sets, Map and SortedMap has Guava utility class Maps and Queue has corresponding utility class Queues. Guava has its own specific interfaces for Multiset with corresponding utility class Multisets, Multimap with utility class Multimaps, BiMap with utility class Maps and Table with utility class Tables.

Utility classes of JCF interfaces are well describing in table on official site [12]:

Guava adds functional programming with Functions and Predicates for JDK 5 and higher even though it is added in plain Java in JDK 8 with Lambda operations.

Furthermore Guavas **hashing** function provides hash using powerful algorithms with possible cryptography, eg. md5 or different hash sizes of algorithm sha.

As cited from official site [13]

[10] `https://docs.oracle.com/javase/8/docs/technotes/guides/collections/changes8.html`

[11] https://github.com/google/guava/wiki/PrimitivesExplained

[12] https://github.com/google/guava/wiki/CollectionUtilitiesExplained

[13] https://github.com/google/guava/#learn-about-guava

Table 4.3: Utility Classes in Guava

Interface	JDK or Guava?	Corresponding Guava utility class
Collection	JDK	Collections2
List	JDK	Lists
Set	JDK	Sets
Map	JDK	Maps
Queue	JDK	Queues
Multiset	Guava	Multisets
Multimap	Guava	Multimaps
BiMap	Guava	Maps
Table	Guava	Tables

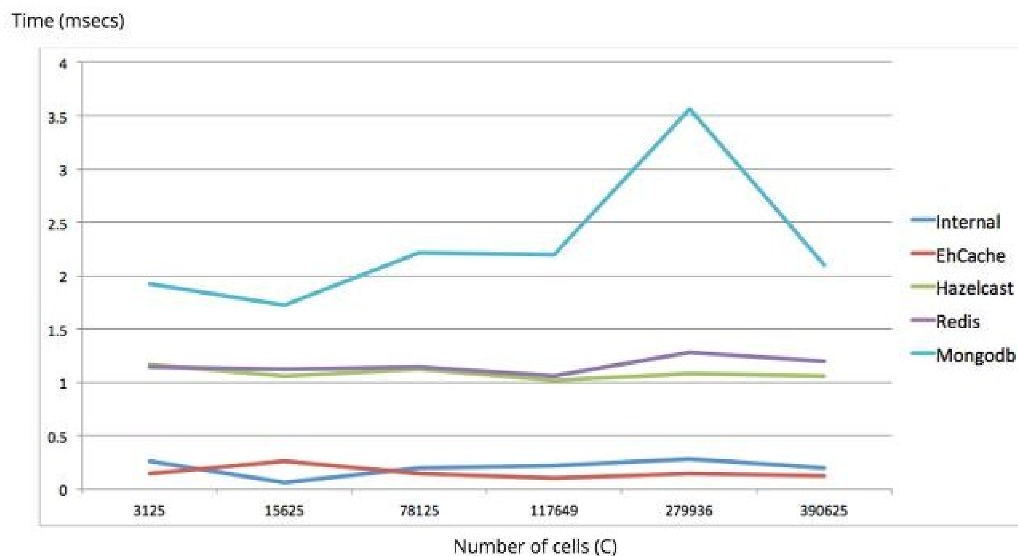
Guava provides many more utilities along these lines: static methods applicable to all collections.

4.3 Distributed Caches

To process large data it is necessary to use distributed caching frameworks. Nowadays there are multiple famous projects: EHcache, Hazelcast, Hadoop, Redis and Mongodb.

I would like to gain hands on experience and produce comparison like in the following picture (Figure 4.1).[14]

Figure 4.1: Comparison of Multiple Cache Frameworks



[14] https://blog.nomissolutions.com/labs/2015/03/10/evaluation-of-caching-frameworks/

5 Analysis and Assesment of Test Cases

As described in the Efficiency of Algorithms (Section 2.5, page 8) there are two commonly used processes to describe time complexity of algorithm. The theoretical complexity described by asymptotic orders of growth and the practical measuring of time for operations. In this thesis there will be descriptions of algorithms with counted asymptotic orders of growth for each of them and as well time measured operations.

5.1 Microbenchmarking Framework

To solve the issue with **heating up** the CPU before and in time of the testing in this bachelor thesis I used framework Java Microbenchmarking Harness (JMH). There is huge number of possible settings from warm up, multiple forks, measurements to possible outputs. I will describe main settings for JMH that I will use during testing.

Main setting is Benchmark with annotation @Benchmark it is above the method that should be benchmarked. The Warm up uses annotation @Warmup with few possible parameters like number of iterations, number of time spent, time units and batch size. The Measurement has annotation @Measurement and the same set of possible parameters. The Fork has annotation @Fork with possible parameters number of warmup forks where are results ignores, value where are results saved. The Output time unit has annotation @OutputTimeUnit where is set the enum value from TimeUnit for all the tests I used TimeUnit.NANOSECONDS. The last setting is Benchmark mode with annotation @BenchmarkMode that sets enum value of Mode what the Benchmarking should output in this bachelor thesis I used Mode.AverageTime.

5.1.1 Starting Test Cases

Note that both the output and the results may look different on other machine and therefore the test cases will be tested on various computer configurations as well as on various operation systems. To use test cases created in this bachelor thesis it is possible to run the codes attached. To run test cases you need JVM 8 and Maven 3. There is either possibility to run tests in console or in the IDEs.

Console: Open project folder and write this code. Note: first line depends on installation and settings of Maven, if is installed only Maven 3 it would be called **mvn install**. First line compiles and stores the application in local repository. Second line starts testing of test cases in console.

Listing 5.1: Console triggered test cases

```
mvn3 install
java -jar target/benchmarks.jar -f 5 -i 10 -wi 10
```

To not fork as often and don't do so many measure runs (for demo purposes), you can apply the following flags: **-i 5 -f 1 -wi 5**. This will only trigger one fork and do 5 iterations for warming up and measuring as well. There are dozens of flags you can tune to your needs.

5.2 Analysis of Test Cases

The test cases will be separated for each type of collection and operation. The implementations of the same types of collection and operation will be compared. The results will be represented by table of resulting operations per second or time to complete specified amount of operations and graph which be showing differences in particular implementations.

5.2.1 Construction Operation

As multiple collection frameworks use special construction of the collections it is interesting to find out if the implementations are adding some value, speeding up or slowing down and if they.

The test will consist of only constructions. In the tests there will be description of the changes, additional code will be inspected and there will be graphs of time for process.

5.2.2 Insert Operation

As mentioned in Comparability (Section 2.3, page 4) in any collection the insertion of element is provided in special way. Some collections use hash table where is the result set in, some use linking nodes where adding last is always in constant time and some use tree structures where is needed balancing as well as comparing. Those implementations can result in huge differences in time consumed whereas giving special functionality like sorting collection.

It will be important to test insertion in beginning of collection, in the middle of collection and in the end of collection.

5.2.3 Get Operations

If there is any purpose of storing the data in collection it is to retrieve objects. To know what object should be retrieved it is important to know index, key or iterator. Every operation of those described will be tested for every collection that implements method for this operation.

5.2.4 Remove Operation

As in case of an insertion there are different implementations of removing the elements from collection. In some implementations will be special operations inside remove operation like re-linking or re-balancing. It will be important to test remove operations in beginning, middle and in the end of collection.

5.2.5 Sort Operation

Sort operation is not inside the collections but inside utility classes. As there is big number of sorting algorithms it is interesting to test sorting of all types of collections and as well comparing how well the sorted and presorted collections will stand. If presorted collections, like trees, are better then sorted-at-time collections.

6 Design of Test Cases

Every collection structure described in Implementations of Data Structures (Section 3.1, page 13), their counterparts (eg. HPPC ObjectObjectOpenHashMap<Key, Value>) from other collections and additional collections (eg. Google Guava MultiSet, or MultiMap) will be measured and the resulting benchmarks will be demonstrated with tables and graphs.

6.1 Division of Test Cases into Operations

Test cases will be structured into operations as in Analysis of Test Cases (Section 5.2, page 29). Each of the operations will be tested in basic and special cases, which will show structures advantages and disadvantages from the perspective of other collections.

6.1.1 Test Cases – Insert Operations Design

To use collections it is necessary to insert elements. Insertion operation is provided in collections using methods *add/put*. Method *add* is in all collection types that implement *Collection* interface. The only exception is Map interface. Map implementations instead do implement *put* method which consists of key value association.

In case of insertion there are often provided two possibilities to insert element(s) into collection. With, or without providing index. Providing index means adding the element in the index place which cannot exceed the size of the collection. This affects only ordered collections.

Adding element inside collection without index is provided for all collection types from JCF. The exception of the collection included in Java in which inserting into collection has to be using index is array but it doesn't belong to JCF.

As written above basic collections implement insertion without index. My hypothesis is that insertion without index should provide same or better results than adding elements with specified index.

Insert operation will be divided in multiple areas:

- Inserting primitive values, if not possible inserting objects that auto-boxed primitive values.
- Inserting objects created with some parameters.
- Inserting elements without index. (Applies to first two areas)
- Inserting elements in specified index if possible. (Applies to first two areas)

6.1.2 Test Cases – Get Operations Design

Using collections has most of the time one reason – to read what was added or compare the values. Get operation is provided in List interface with parameter index, Map interface has get operation with parameter of Generic type and in Set interface there is no get operation.

Next operation that I consider underneath get operation is iterator. All of the Collection interface implementations provide iterator to iterate over collection. Maps have entrySet, keySet and values that are able to be iterated over because all of the returned collection structures are of Collection interface and therefore implement iterator. The iteration over the collection is ordered only if the collection itself is ordered.

Another operation that I consider underneath get operation is contains operation. Contains operation searches for generic type object which was provided as parameter and returns true if

object was found in collection, otherwise false, which is of type boolean. Operation contains is provided by all collections with only exception Map interface that provides containsKey, with parameter of generic type of map.key, and containsValue, with parameter of generic type of map.value, with same return type as basic contains method.

Get operation will be divided in multiple areas:

- Getting primitive values, if not possible getting objects that auto-boxed primitive values.
- Getting objects created with some parameters.
- Iterating over collections that implement iterator method.
- Using contains elements, for maps containsKey and containsValue.

6.1.3 Test Cases – Remove Operations Design

Remove operation is contained in every collection from JCF with parameter generic type object. Special cases are in ordered collections from List and Queue interfaces where it is possible to remove objects by index and other special case is in Map interface via handing as parameters generic type object-key and generic type object-value to remove method.

Remove operation will be divided in multiple areas:

- Removing primitive values, if not possible removing objects that auto-boxed primitive values.
- Removing objects created with some parameters.
- Removing elements without index if possible. (Applies to first two areas)
- Removing elements in specified index if possible. (Applies to first two areas)

7 Java Collections Framework – Basic Collections

In this chapter will be implementation and demonstration by benchmarks and graphs of Java Collections Framework in JDK 7 and the JDK 8 of basic collection types described in Implementations of Data Structures (Section 3.1, page 13).

7.1 Java Collections Framework – Interfaces and Operations

Test cases for Java Collections Framework test every collection operation described in Design of Test Cases (Section 6, page 31). Test cases are created for specific operations of the Java Collections Framework interface and according methods. The tested methods are described in Table 7.1.

Table 7.1: Table JCF Interfaces and Tested Operations

	Insert	**Get**	**Remove**
List ArrayList LinkedList	add() add(i) add(0)	get(i) contains(o)	remove(i) remove(o)
Set HashSet TreeSet	add()	contains(o)	remove(o)
Map HashMap TreeMap	put()	get(k) contains(k)	remove(k)

Where i means index, o means object and k means key object (only available in Map interface) and a special case is with add(0) which is inserting as first element.

The methods described in Table 7.1 are shortened to be more informative. Every method in insert bears the object (or object key and object value in Map interface) to be added. These methods are shortened to show if the objects can be added in the specified index place. It was necessary to include threshold values to show main differences in implementations (like value 0 in add in List interface).

7.2 Java Collections Framework – List Interface Performance

List interface in Java Collections Framework has multiple implementations, but the most often used List implementations are ArrayList and LinkedList. These two implementations were tested and compared in this section.

Although both list designs are very good in case of complexity. Each of the list implementation have its advantages and disadvantages. These advantages will be shown in test cases and according graphs.

ArrayList

Insert operation case performance – amortized $\mathcal{O}(1)$
Get operation case performance – $\mathcal{O}(1)$
Remove operation best/worst case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Space complexity – $\mathcal{O}(n)$

LinkedList

Insert operation best/worst case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Get operation case performance – $\mathcal{O}(n)$
Remove operation best/worst case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Space complexity – $\mathcal{O}(n)$

7.2.1 Insert Operation in List Interface

ArrayLists complexity is amortized $\mathcal{O}(1)$ (2.5.3). The amortization is important, because of occasional increase of array size, which is $\mathcal{O}(n)$ that is needed to move all n elements to the newly created bigger array.

LinkedLists complexity is somewhere between 1 and $n/2$. The basic addition of element after last node in LinkedList or before first node is 1 as both sides of the LinkedList are the starting nodes. Adding the element in the middle has a worst case scenario, that is $n/2$, because it is needed to iterate over nodes to get in the place. It is 'only' $n/2$, because of two possible starting points at the beginning or the end of the LinkedList. If the index is closer to the end iteration starts with last node, else iteration starts with first node.

7.2.1.a Insert Operations in List Interface

Test cases of Insert operation in List interface consist of methods that are described in Table 7.1.

The methods described in Table 7.1 are shortened to be more informative. Shortened names of methods will be kept to be referenced in sections.

The shortened methods are add(), add(i) and add(0), where i is a index. While method names are shortened they point to real methods like add() is add(Object), add(i) is add(index, Object), with special case at zero index value add(0), add(0,Object).

7.2.2 Insert Operation – Operation add() in List Interface

The insert operation uses add() method based on Collection interface.

To insert all elements inside the specific list implementation I have created method fillList. This method (Listing 7.1) iterates over array of generic objects and adds every object in the List implementation.

Listing 7.1: Method fillList inserts elements into List implementation

```
<T> void fillList(List<T> list, T[] objectArray, int noObjects) {
    for (int i = 0; i < noObjects; i++) {
        list.add(objectArray[i]);
    }
}
```

This operation is supposed to be amortized $\mathcal{O}(1)$ for ArrayList and $\mathcal{O}(1)$ for LinkedList.

7.2.2.a Operation add() in List Interface – Graphic Results

Figure 7.1: Java 7 Operation add() on Lists – Double

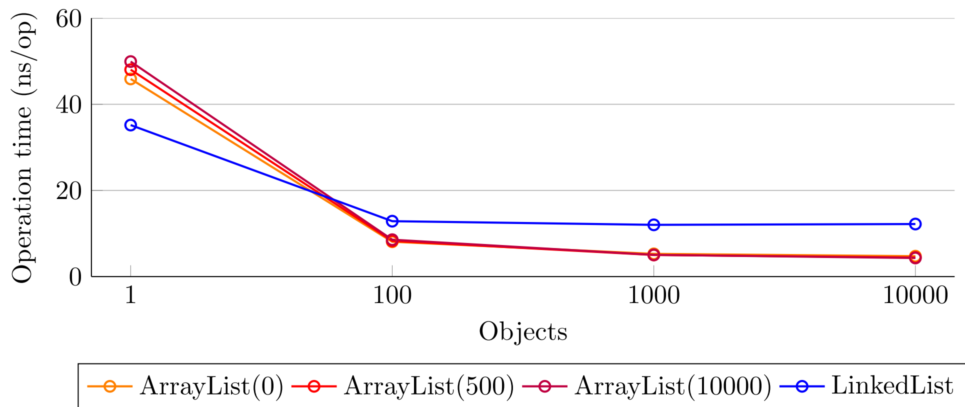


Figure 7.2: Java 8 Operation add() on Lists – Double

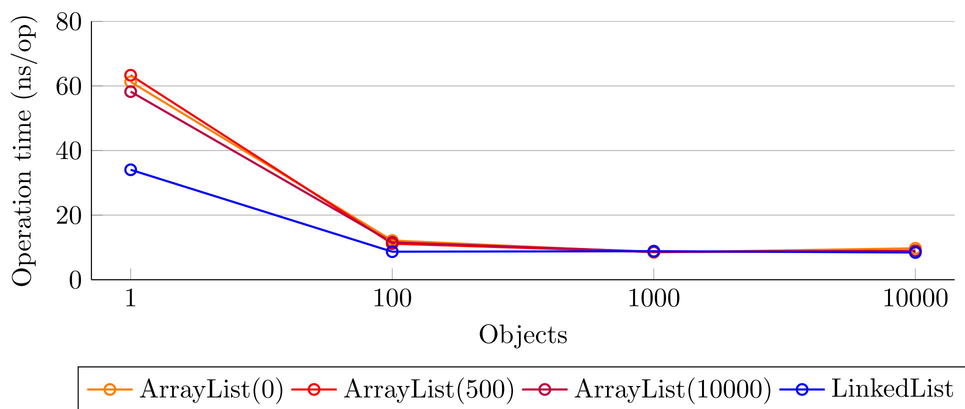


Figure 7.3: Java 7 Operation add() on Lists – MockObject

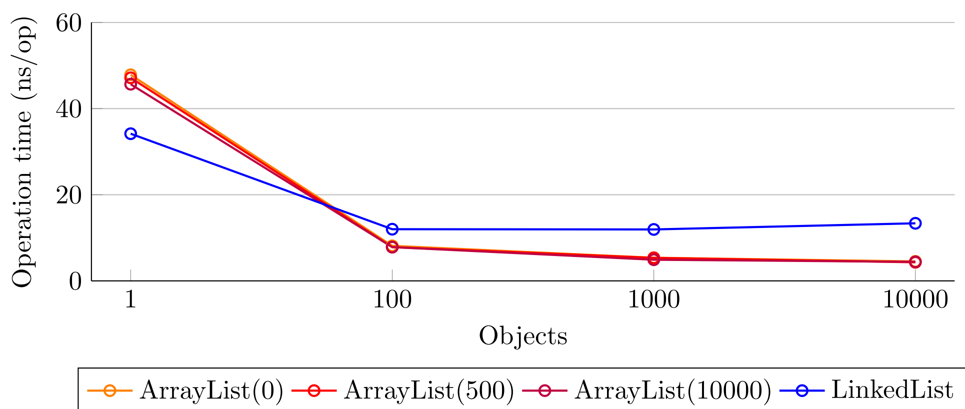


Figure 7.4: Java 8 Operation add() on Lists – MockObject

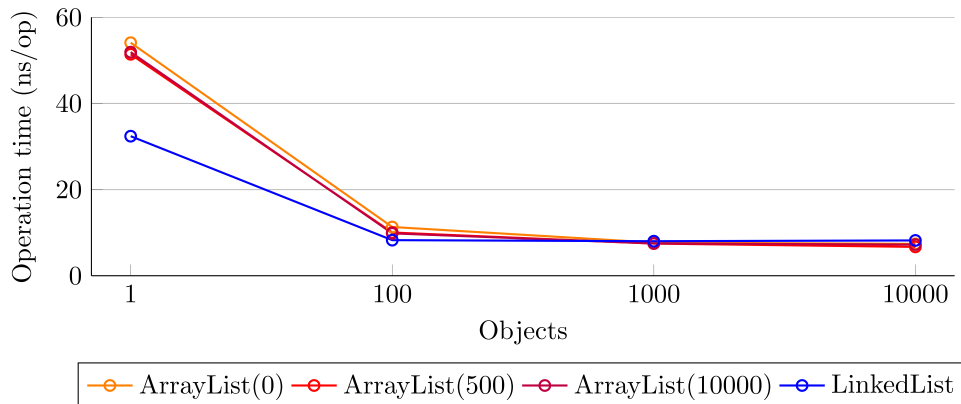


Table 7.2: Average time for add() operation in ArrayList (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – ArrayList	50.95	9.35	6.65	6.28
Java 8 – ArrayList	53.90	9.78	6.89	6.80
Difference	2.95	0.43	0.25	0.51

Table 7.3: Average time for add() operation in LinkedList (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – LinkedList	34.68	12.43	11.98	12.79
Java 8 – LinkedList	33.22	8.47	8.43	8.31
Difference	-1.46	-3.96	-3.55	-4.47

7.2.3 Insert Operation – Operation add(i) in List Interface

The insert operation uses add(i) method based on List interface.

To insert all elements inside the specific list implementation I have created and overloaded method fillList. This method (Listing 7.2) iterates over array of generic objects and adds every object in the List implementation at position of index i.

Listing 7.2: Method fillList inserts elements into List implementation

```
<T> void fillList(List<T> list, T[] objectArray,
                  int[] orderArray, int noObjects) {
    for (int i = 0; i < noObjects; i++) {
        list.add(orderArray[i], objectArray[i]);
    }
}
```

This operation complexity is supposed to be $n - i$ for ArrayList and $n/2$ for LinkedList, where i is index of newly added element.

For specific index values of i (i equal 0, or size of List n) is LinkedLists complexity equal $\mathcal{O}(1)$, whereas index i equal 0 is the worst case of ArrayList with $\mathcal{O}(n)$.

7.2.3.a Operation add(i) in List Interface – Graphic Results

Figure 7.5: Java 7 Operation add(i) on Lists – Double

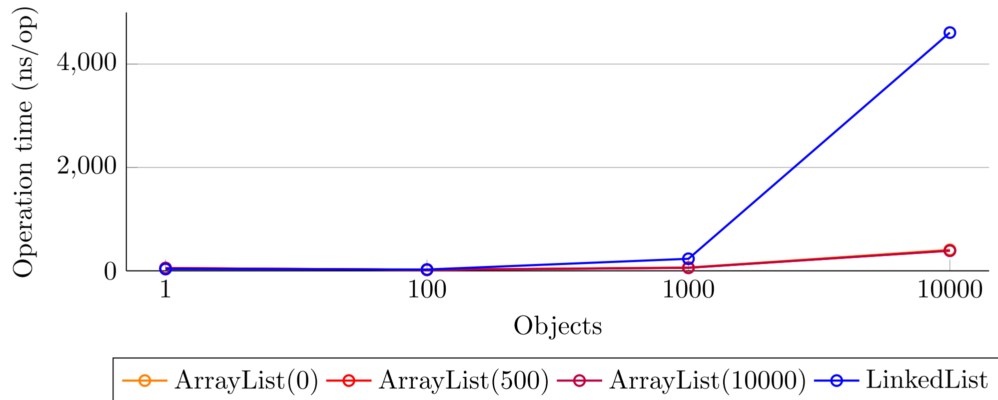


Figure 7.6: Java 8 Operation add(i) on Lists – Double

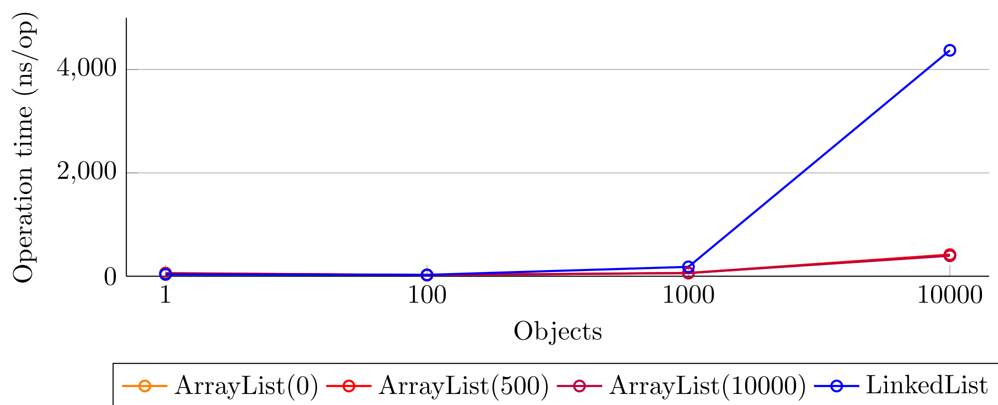


Figure 7.7: Java 7 Operation add(i) on Lists – MockObject

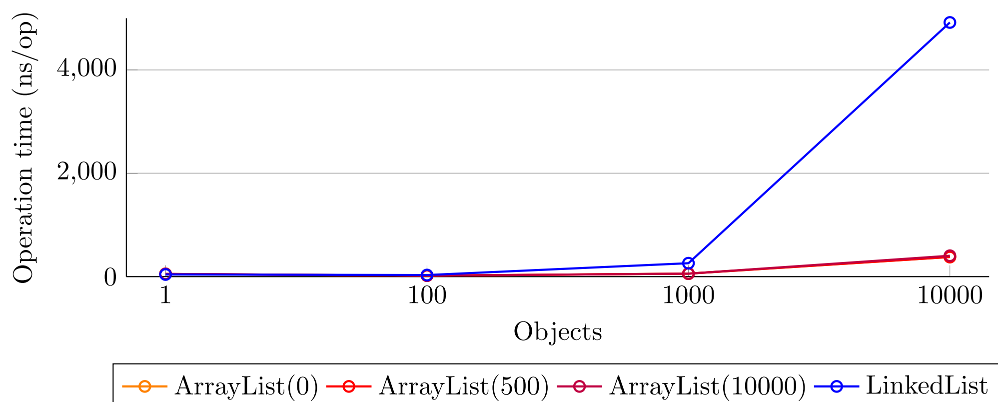


Figure 7.8: Java 8 Operation add(i) on Lists – MockObject

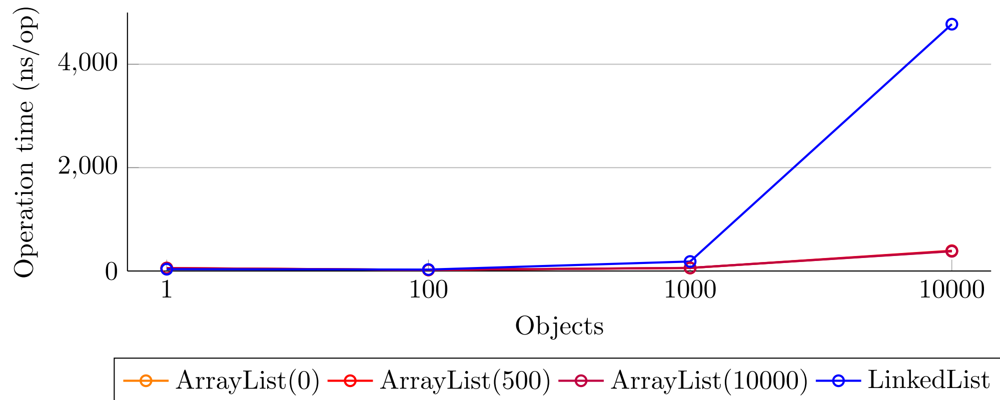


Table 7.4: Average time for add(i) operation in ArrayList (ns/op)

	1	100	1.000	10.000
Java 7 – ArrayList	53.62	19.98	61.13	395.93
Java 8 – ArrayList	58.39	23.25	62.64	410.67
Difference	4.77	3.27	1.51	14.75

Table 7.5: Average time for add(i) operation in LinkedList (ns/op)

	1	100	1.000	10.000
Java 7 – LinkedList	34.72	25.88	233.28	4 609.51
Java 8 – LinkedList	34.70	27.14	181.90	4 370.99
Difference	-0.02	-1.26	-51.38	-238.52

7.2.4 Insert Operation – Operation add(0) in List Interface

The insert operation uses add(i) method based on List interface.

To insert all elements inside the specific list implementation I have created method fillFirstInList. This method (Listing 7.3) iterates over array of generic objects and adds every object in the List implementation at the start of list.

Listing 7.3: Method fillFirstInList inserts elements into List implementation

```
<T> void fillFirstInList(List<T> listToFill,
                                T[] fillingArray, int noObjects) {
    for (int i = 0; i < noObjects; i++) {
        listToFill.add(0, fillingArray[i]);
    }
}
```

This operation complexity is supposed to be $\mathcal{O}(n)$ for ArrayList and $\mathcal{O}(1)$ for LinkedList. This means it is the worst case scenario for ArrayList and best case scenario for LinkedList.

7.2.4.a Operation add(0) in List Interface – Graphic Results

Figure 7.9: Java 7 Operation add(0) on Lists – Double

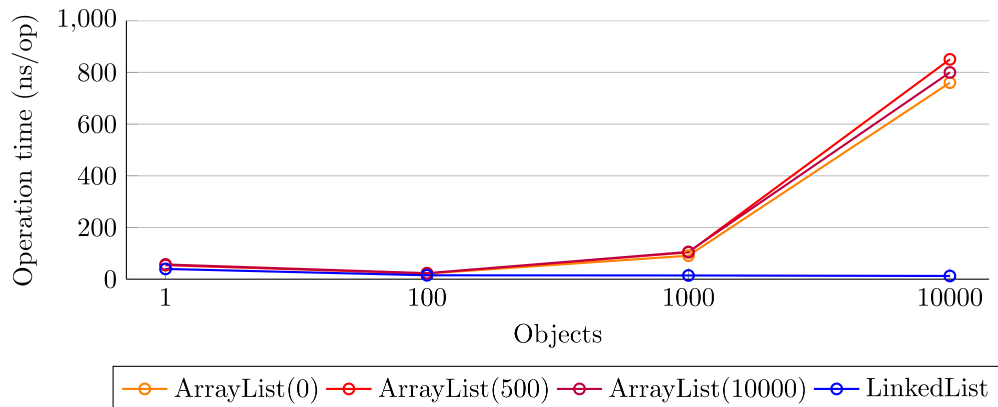


Figure 7.10: Java 8 Operation add(0) on Lists – Double

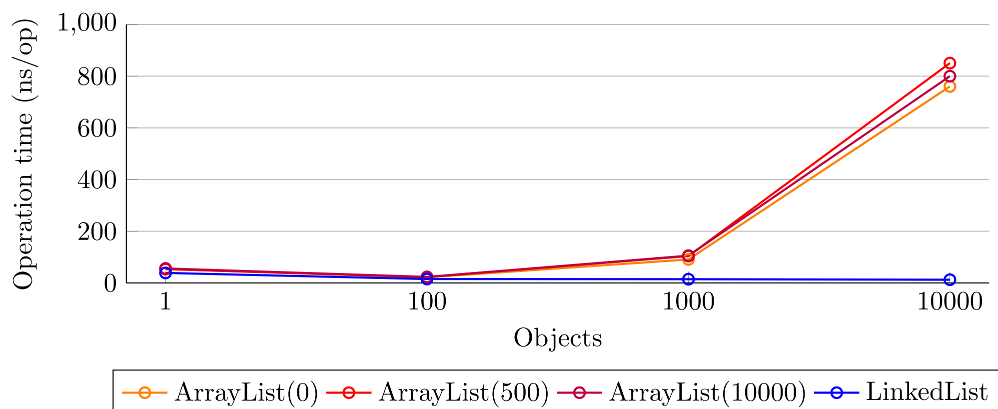


Figure 7.11: Java 7 Operation add(0) on Lists – MockObject

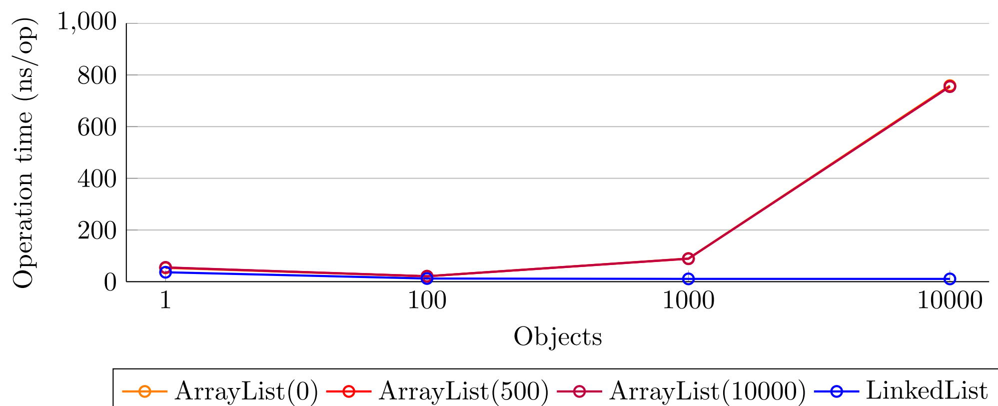


Figure 7.12: Java 8 Operation add(0) on Lists – MockObject

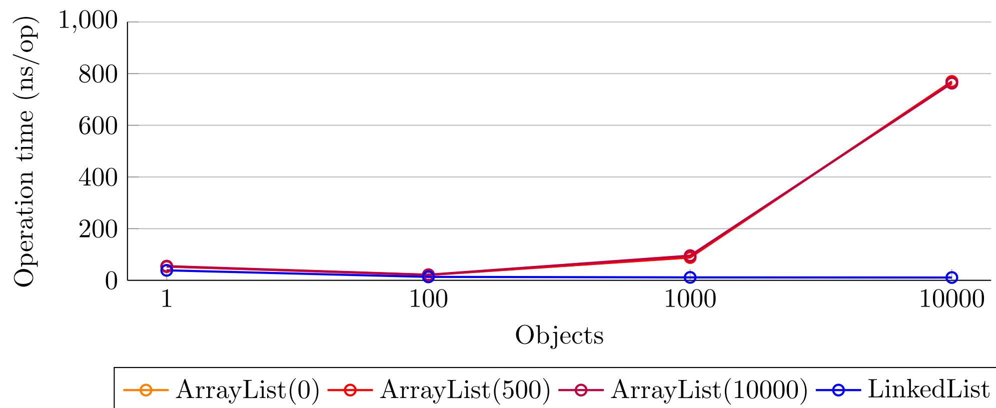


Table 7.6: Average time for add(0) operation in ArrayList (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – ArrayList	57.00	22.64	95.17	776.13
Java 8 – ArrayList	54.13	21.88	95.74	785.31
Difference	-2.87	-0.76	0.57	9.18

Table 7.7: Average time for add(0) operation in LinkedList (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – LinkedList	41.55	14.41	13.39	12.67
Java 8 – LinkedList	38.70	14.25	12.91	11.70
Difference	-2.86	-0.16	-0.47	-0.97

7.2.5 Operation Insert on Lists in Java Collections Framework – Findings

First interesting finding and quite obvious from the graphs is that adding objects with multiple variables is as fast as adding auto-boxed primitive types. It has basically no difference (time difference between adding Object and adding Double is oscillating for each case differently, eg. for method add() difference was $\pm 0.7\ ns/op$).

Second finding is, that the ArrayList has roughly the same time complexity for pre-set capacity parameter (based on the three sizes from experiment benchmarking – 0, 500 and 10000). The biggest difference is in adding only one element where the difference is in time spent pre-allocating the buffer with the specified size.

Third finding is, that adding elements in basic order is faster for ArrayList than in LinkedList variant (except the adding first element). The average time difference for adding 100, 1.000 and 10.000 elements is $\pm 6.5\ ns$. Java 8 had slightly improved LinkedLists performance and therefore it is almost as fast as ArrayList.

Fourth finding is, that add operations time complexity $\mathcal{O}(1)$ for ArrayList is faster on average than for LinkedList ($\pm 2.1\ ns/op$).

Fifth finding is, that hypothesis of theoretical complexities of both ArrayList and LinkedList for add operations are (close to being) correct. Therefore it is possible to accept this hypothesis.

7.2.6 Get Operation in List Interface

ArrayLists complexity of method get() is $\mathcal{O}(1)$, whereas contains() runs in constant time $\mathcal{O}(n)$.

LinkedLists complexity of method get() is $n/2$ with best case equal $\mathcal{O}(1)$. The contains() runs in $\mathcal{O}(n)$.

7.2.7 Get Operation – Operation get(index) in List Interface

The get operation uses get() method based on List interface.

To fetch elements from the list implementation I have created method getRandomFromList. This method (Listing 7.4) iterates over list and randomly fetches objects.

Listing 7.4: Method getRandomFromList fetches elements from List implementation

```
<T> void getRandomFromList(List<T> filledList, int[] order,
                           int noObjects, Blackhole blackhole) {
        for (int i = 0; i < noObjects; i++) {
            blackhole.consume(filledList.get(order[i]));
        }
}
```

This operation is supposed to be equal $\mathcal{O}(1)$ for ArrayList and $n/2$ for LinkedList.

7.2.7.a Operation get(index) in List Interface – Graphic Results

Figure 7.13: Java 7 Operation get(index) in Lists

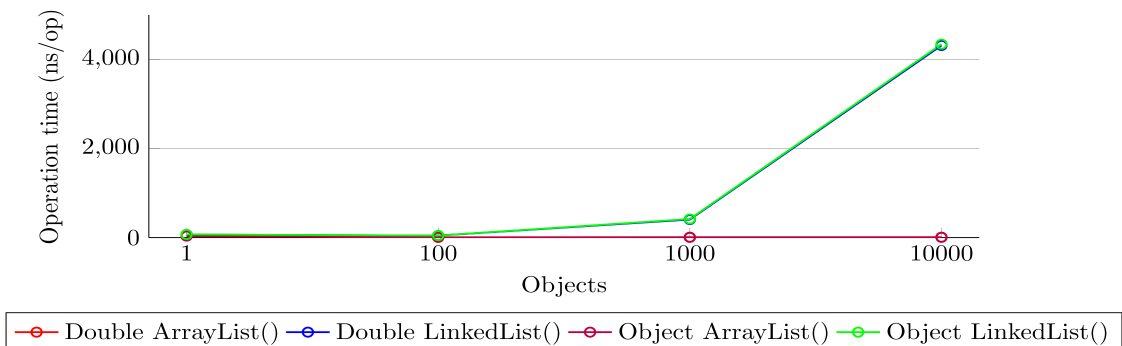


Figure 7.14: Java 8 Operation get(index) in Lists

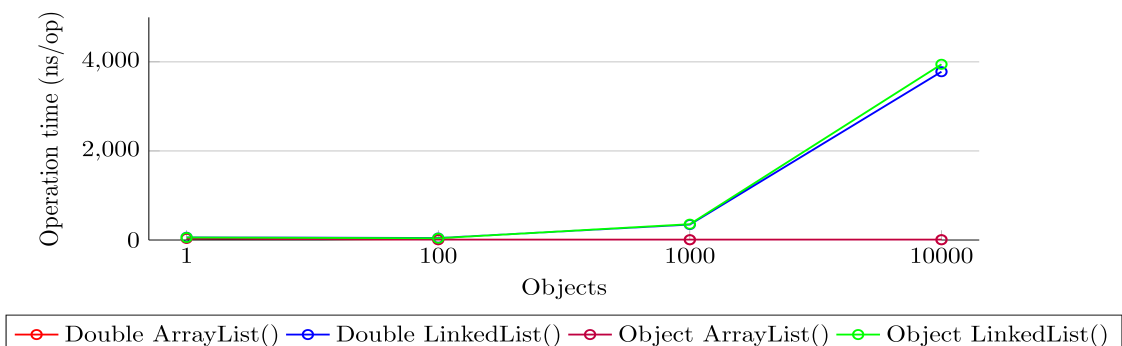


Table 7.8: Average time for get(index) operation in ArrayList (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – ArrayList	35.47	5.86	7.23	7.50
Java 8 – ArrayList	37.46	6.59	6.44	6.58
Difference	1.99	0.73	-0.80	-0.92

Table 7.9: Average time for get(index) operation in LinkedList (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – LinkedList	68.87	44.99	411.54	4328.96
Java 8 – LinkedList	59.41	42.83	351.30	3862.38
Difference	-9.47	-2.16	-60.25	-466.58

7.2.8 Get Operation – Operation contains(Object) in List Interface

The get operation contains(o) method based on Collection interface.

To find out if the specific list implementation contains element I have created method containsInList. This method (Listing 7.5) iterates over array of generic objects and tries objects containing in the List implementation.

Listing 7.5: Method containsInList for List implementation

```
<T> void containsInList(List<T> filledList, int[] order, T[] objects,
                        int noObjects, Blackhole blackhole) {
    for (int i = 0; i < noObjects; i++) {
        blackhole.consume(filledList.contains(objects[order[i]]));
    }
}
```

This operation complexity is $\mathcal{O}(n)$ for ArrayList as well as for LinkedList.

7.2.8.a Operation contains() in List Interface – Graphic Results

Figure 7.15: Java 7 Operation contains() in Lists

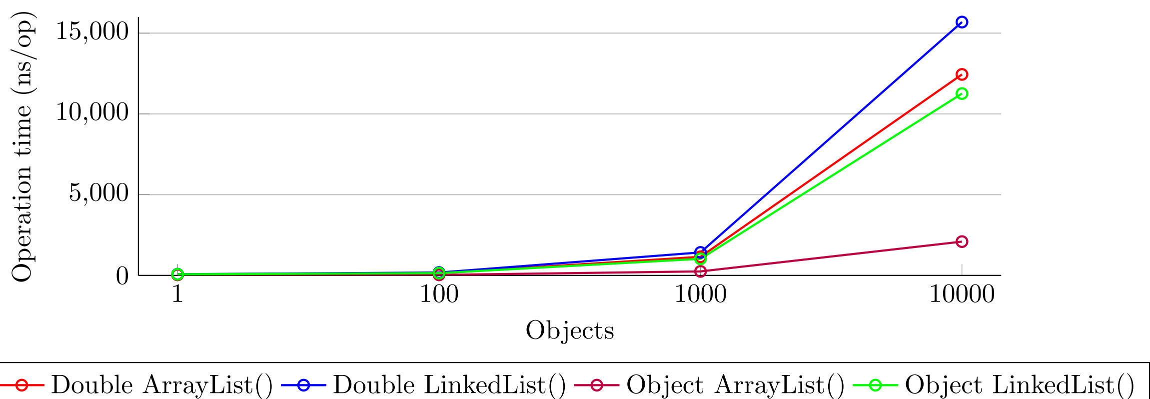


Figure 7.16: Java 8 Operation contains() in Lists

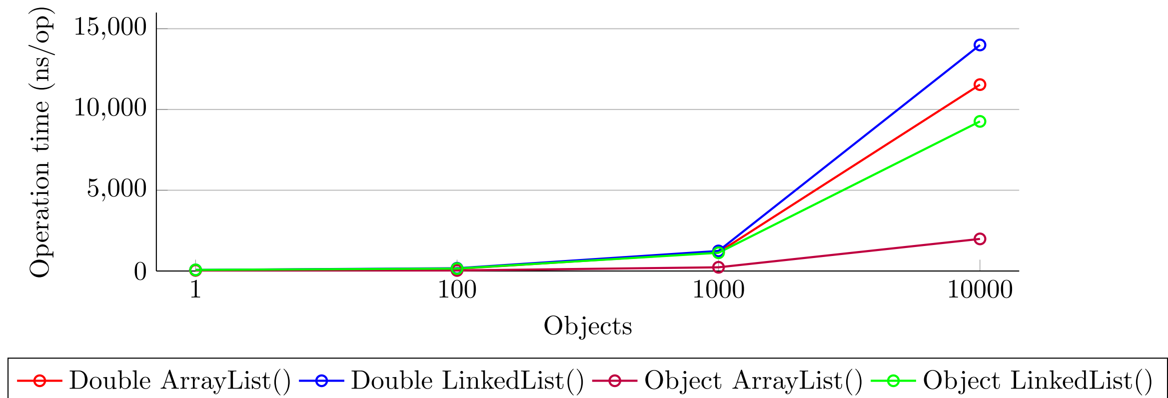


Table 7.10: Average time for contains() operation in ArrayList (ns/op)

	1	100	1.000	10.000
Java 7 – ArrayList	39.01	87.04	700.24	7267.44
Java 8 – ArrayList	42.68	84.53	693.31	6763.17
Difference	3.67	-2.51	-6.93	-504.27

Table 7.11: Average time for contains() operation in LinkedList (ns/op)

	1	100	1.000	10.000
Java 7 – LinkedList	73.00	159.80	1224.94	13470.35
Java 8 – LinkedList	67.44	161.60	1185.56	11628.08
Difference	-5.56	1.81	-39.39	-1842.27

7.2.9 Operation Get on Lists in Java Collections Framework – Findings

First finding is that using get() method to fetch objects with multiple variables is as fast as fetching auto-boxed primitive types. The difference is minor and can be taken as unimportant factor (average difference was ± 6.13 ns/op).

Second finding is by observing it is obvious that using contains() method for objects with multiple variables is much faster than finding auto-boxed primitive type Double. More details later in Java Collections Framework – Results anomaly (Section 7.5, page 60). The objects in get() method are as fast as auto-boxed primitive types.

Third finding is, that hypothesis of theoretical complexities of both ArrayList and LinkedList for get operations are (close to being) correct. Therefore it is possible to accept this hypothesis.

7.2.10 Remove Operation in List Interface

ArrayLists complexity of method remove(index) is equal to $n - i$ while remove(o) is $\mathcal{O}(n)$.

LinkedLists complexity of method remove(index) is somewhere between $n/2$ and 1 while remove(o) is $\mathcal{O}(n)$.

7.2.11 Remove Operation – Operation remove(index) in List Interface

The remove operation uses remove(index) method based on List interface.

To remove all elements inside the specific list implementation I have created method removeRandomInList. This method (Listing 7.6) iterates over array of generic objects and removes object according to specified index from List implementation.

Listing 7.6: Method removeRandomInList removes elements from List implementation

```
<T> void removeRandomInList(List<T> filledList, int[] order,
                            int noObjects, Blackhole blackhole) {
    for (int i = 0; i < noObjects; i++) {
        blackhole.consume(filledList.remove(order[i]));
    }
}
```

This operation is supposed to be $n - i$ for ArrayList and for LinkedList ranging from $n/2$ to 1.

7.2.11.a Operation remove(index) in List Interface – Graphic Results

Figure 7.17: Java 7 Operation remove(index) from Lists

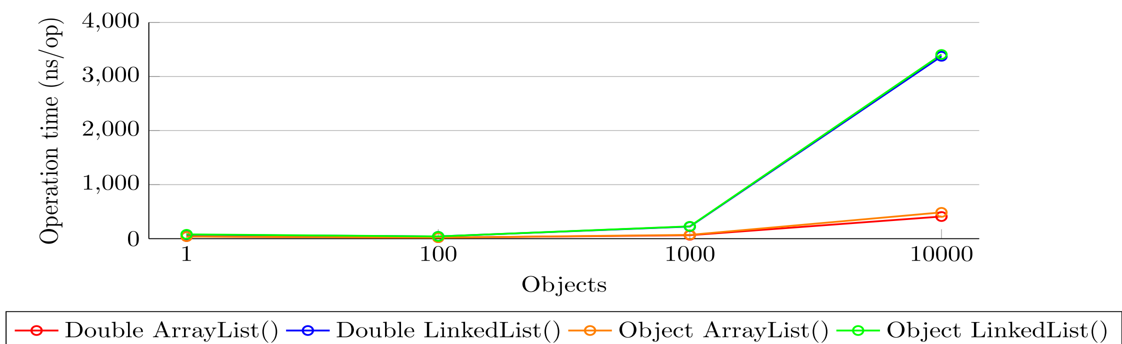


Figure 7.18: Java 8 Operation remove(index) from Lists

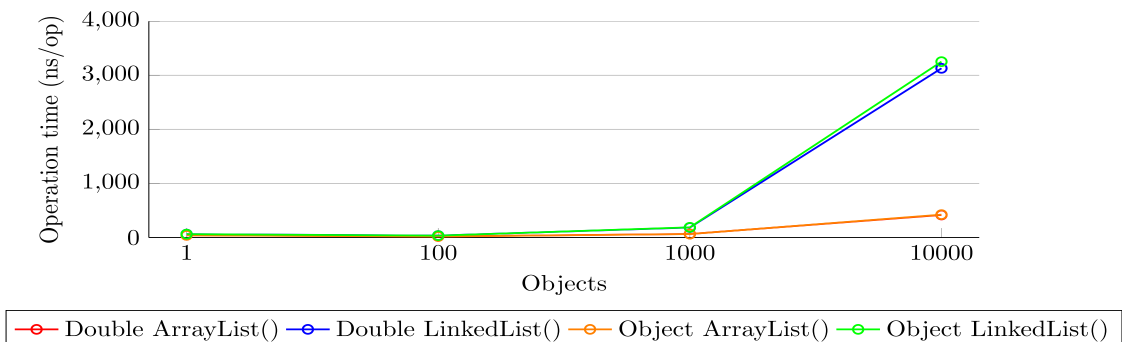


Table 7.12: Average time for remove(index) operation in ArrayList (ns/op)

	1	100	1.000	10.000
Java 7 – ArrayList	39.97	20.07	67.46	448.35
Java 8 – ArrayList	41.69	19.86	64.80	420.29
Difference	1.72	-0.21	-2.67	-28.09

Table 7.13: Average time for remove(index) operation in LinkedList (ns/op)

	1	100	1.000	10.000
Java 7 – LinkedList	76.68	42.01	226.57	3390.37
Java 8 – LinkedList	62.63	35.86	186.80	3191.43
Difference	-14.05	-6.15	-39.77	-198.95

7.2.12 Remove Operation – Operation remove(Object) in List Interface

The remove operation uses remove(i) method based on List interface.

To remove elements inside the specific list implementation I have created method removeRandomInListByObject. This method (Listing 7.1) iterates over array of generic objects and removes every object from the List implementation.

Listing 7.7: Method removeRandomInListByObj removes elements from List implementation

```
<T> void removeRandomInListByObj(List<T> filledList, int[] order,
                T[] objects, int noObjects, Blackhole blackhole) {
    for (int i = 0; i < noObjects; i++) {
        blackhole.consume(filledList.remove(objects[order[i]]));
    }
}
```

This operation complexity is supposed to be $\mathcal{O}(n)$ for both ArrayList and LinkedList.

7.2.12.a Operation remove(Object) in List Interface – Graphic Results

Figure 7.19: Java 7 Operation remove(Object) on Lists

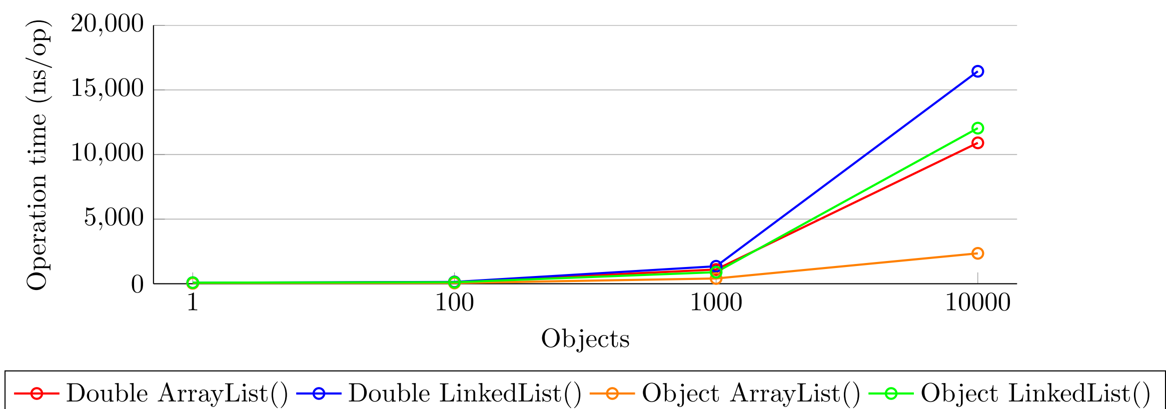


Figure 7.20: Java 8 Operation remove(Object) on Lists

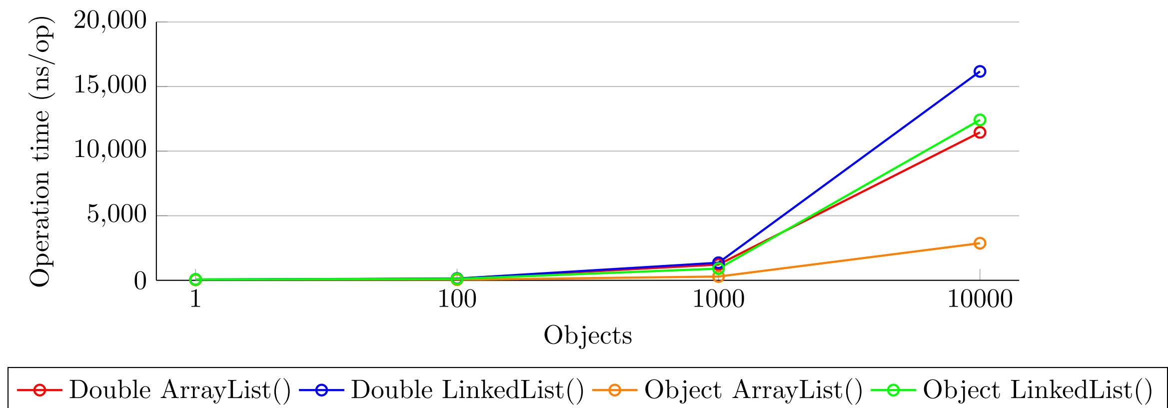


Table 7.14: Average time for remove(Object) operation in ArrayList (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – ArrayList	44.24	94.17	755.53	6628.57
Java 8 – ArrayList	42.57	91.28	758.01	7163.16
Difference	-1.67	-2.90	2.48	-534.58

Table 7.15: Average time for remove(Object) operation in LinkedList (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – LinkedList	80.92	128.61	1124.98	14241.02
Java 8 – LinkedList	66.74	127.94	1136.71	14290.81
Difference	-14.18	-0.68	11.73	-49.79

7.2.13 Operation Remove on Lists in Java Collections Framework – Findings

First finding is that using remove(index) method to remove objects with multiple variables is as fast as removing auto-boxed primitive types. The difference is minor and can be taken as unimportant factor (average difference was $\pm 8.29\ ns/op$).

Second finding is by observing it is obvious that using remove(Object) method for objects with multiple variables is much faster than removing auto-boxed primitive type Double. More details later in Java Collections Framework – Results anomaly (Section 7.5, page 60). The objects in remove(index) method are as fast as auto-boxed primitive types.

Third finding is, that hypothesis of theoretical complexities of both ArrayList and LinkedList for remove operations are (close to being) correct. Therefore it is possible to accept this hypothesis.

7.3 Java Collections Framework – Set Interface Performance

Set interface in Java Collections Framework has multiple implementations, but the most often used Set implementations are HashSet and TreeSet which are being tested in this section.

HashSet

Insert operation best/worst case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Get operation case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Remove operation case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Space complexity – $\mathcal{O}(n)$

TreeSet

Insert operation case performance – $\mathcal{O}(log(n))$
Get operation case performance – $\mathcal{O}(log(n))$
Remove operation case performance – $\mathcal{O}(log(n))$
Space complexity – $\mathcal{O}(n)$

7.3.1 Insert Operation – Operation add() in Set Interface

The insert operation uses add() method based on Collection interface.

To insert all elements inside the set implementation I created method fillSet. This method (Listing 7.8) iterates over array of generic objects and adds every object into the set.

Listing 7.8: Method fillSet inserts elements into Set implementation

```
<T> void fillSet(Set<T> setToFill, T[] fillingArrayKeys,
                                int noObjects) {
    for (int i = 0; i < noObjects; i++) {
        setToFill.add(fillingArrayKeys[i]);
    }
}
```

This operation is supposed to be $\mathcal{O}(1)$ in case of good hash function and $\mathcal{O}(n)$ for constant hash in HashSet and $\mathcal{O}(log(n))$ for TreeSet.

7.3.1.a Operation add() in Set Interface – Graphic Results

Figure 7.21: Java 7 Operation add() on Sets – Double

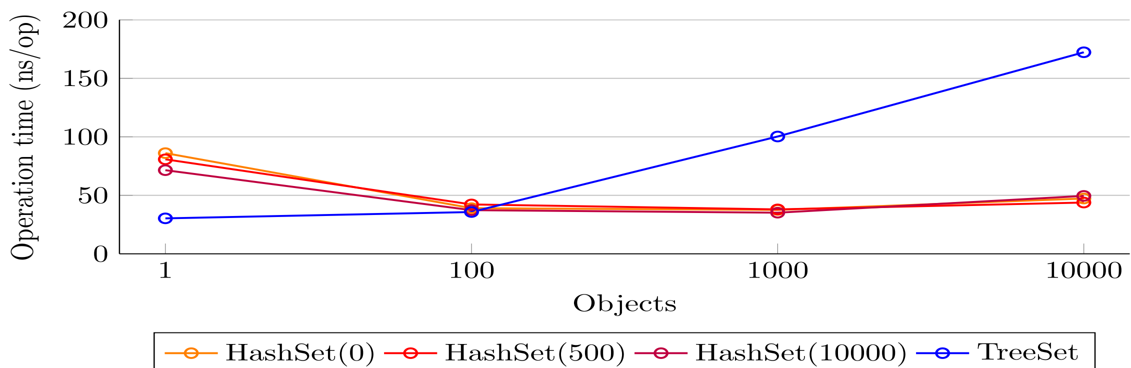


Figure 7.22: Java 8 Operation add() on Sets – Double

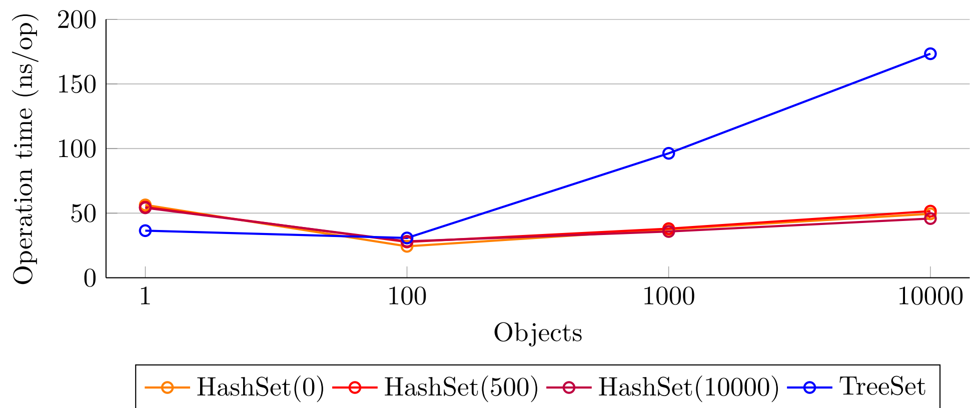


Figure 7.23: Java 7 Operation add() on Sets – MockObject

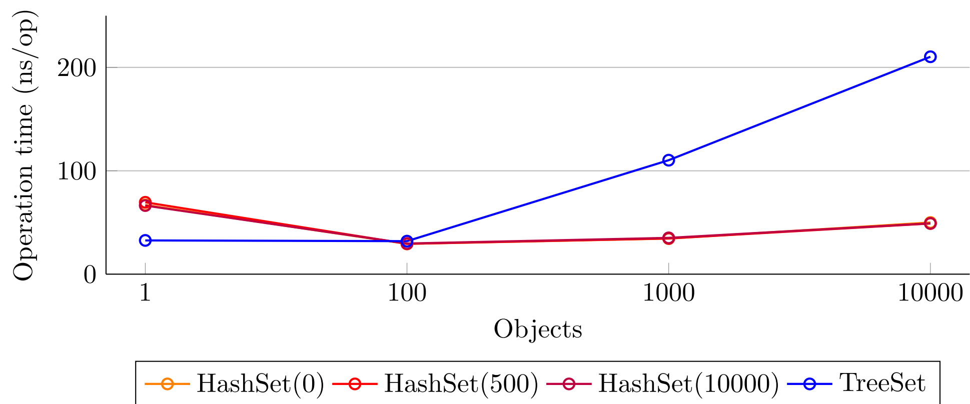


Figure 7.24: Java 8 Operation add() on Sets – MockObject

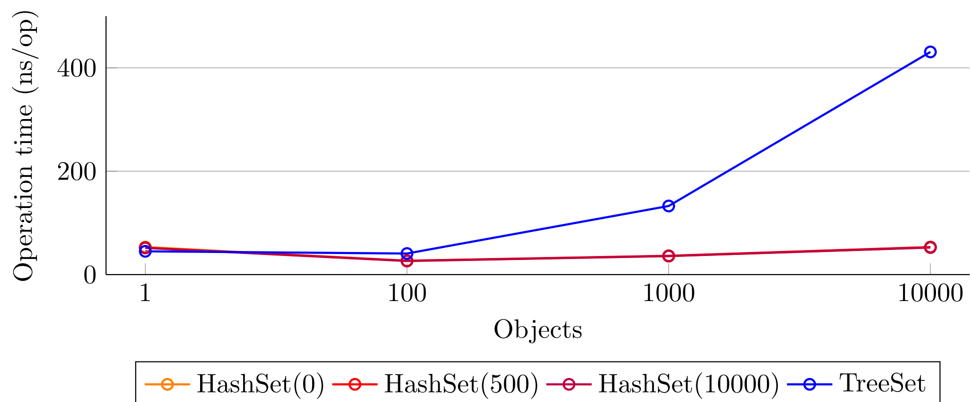


Table 7.16: Average time for add() operation in HashSet (ns/op)

	1	100	1.000	10.000
Java 7 – HashSet	73.53	34.49	35.80	48.13
Java 8 – HashSet	53.70	26.67	36.55	50.88
Difference	-19.83	-7.82	0.75	2.74

Table 7.17: Average time for add() operation in TreeSet (ns/op)

	1	100	1.000	10.000
Java 7 – TreeSet	31.48	33.84	105.27	191.29
Java 8 – TreeSet	40.73	35.78	114.52	302.06
Difference	9.25	1.94	9.25	110.77

7.3.2 Operation Insert on Sets in Java Collections Framework – Findings

First interesting finding and quite obvious from the graphs is that adding objects with multiple variables is almost as fast as adding auto-boxed primitive types. It has some difference (time difference between adding Object and adding Double is oscillating – method add() average difference was ± 92.3 ns/op because of the bad case in Java 8 for 10000 objects. Without this extreme value it is ± 31.7 ns/op).

Second finding is, that the HashSet has roughly the same time complexity for pre-set capacity parameter (based on the three sizes from experiment benchmarking – 0, 500 and 10000). The biggest difference is in adding only one element where the difference is in time spent pre-allocating the buffer with the specified size.

Third finding is, that hypothesis of theoretical complexities of both HashSet and TreeSet for add operations are (close to being) correct. Therefore it is possible to accept this hypothesis.

7.3.3 Get Operation – Operation contains() in Set Interface

The get operation uses contains() method based on Collection interface.

To get elements from the specific set implementation I have created method containsInSet. This method (Listing 7.9) iterates over array of generic objects and tries objects containing in the Set implementation.

Listing 7.9: Method containsInSet for Set implementation

```
<T> void containsInSet(Set<T> filledSet, T[] objects, int[] order,
                                int noObjects, Blackhole blackhole) {
    for (int i = 0; i < noObjects; i++) {
        blackhole.consume(filledSet.contains(objects[order[i]]));
    }
}
```

This operation is supposed to be $\mathcal{O}(1)$ in case of good hash function and $\mathcal{O}(n)$ for constant hash in HashSet and $\mathcal{O}(log(n))$ for TreeSet.

7.3.3.a Operation contains() in Set Interface – Graphic results

Figure 7.25: Java 7 Operation contains() on Set

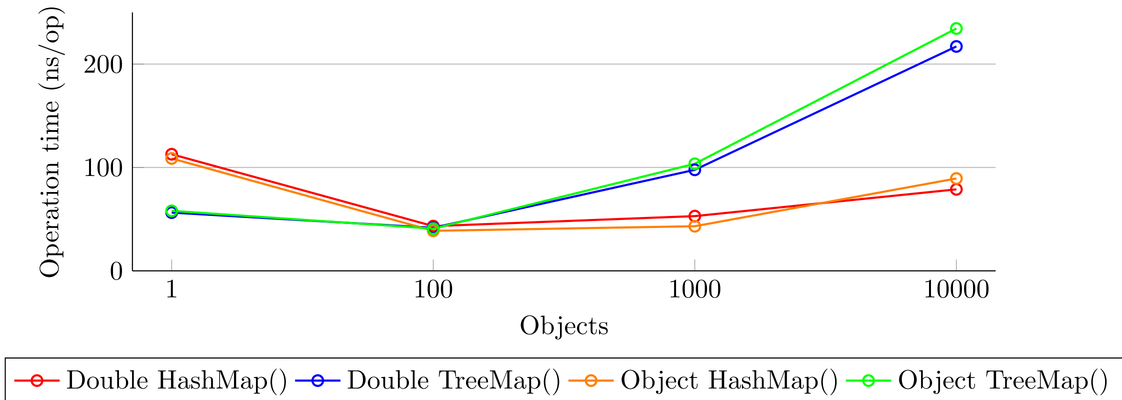


Figure 7.26: Java 8 Operation contains() on Set

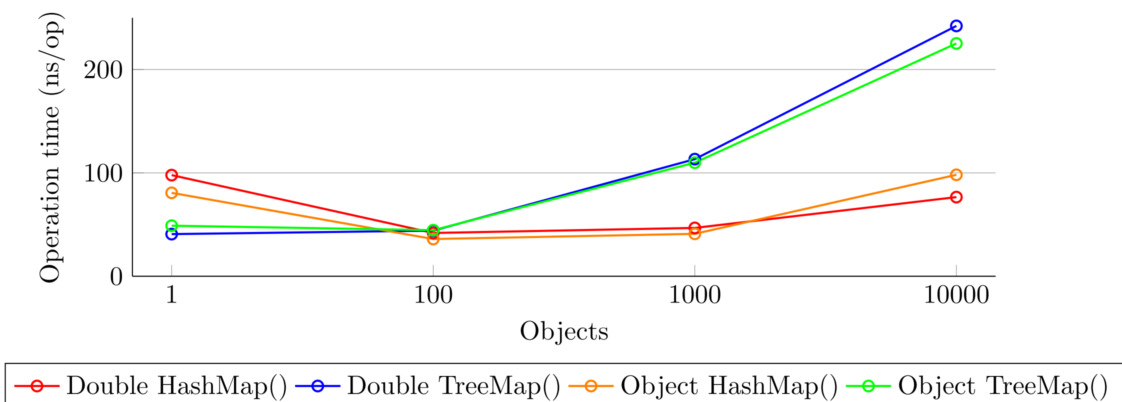


Table 7.18: Average time for contains() operation in HashSet (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – HashSet	110.77	40.99	48.05	84.09
Java 8 – HashSet	89.26	38.94	43.90	87.35
Difference	-21.50	-2.05	-4.15	3.26

Table 7.19: Average time for contains() operation in TreeSet (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – TreeSet	53.07	41.90	105.27	234.11
Java 8 – TreeSet	48.56	44.95	114.53	229.61
Difference	-4.49	3.05	3.40	-4.50

7.3.4 Operation Get on Sets in Java Collections Framework – Findings

First interesting finding from the graphs is that getting objects with multiple variables is on average as fast as getting auto-boxed primitive types. It has basically no difference (time difference between getting Object and Double is oscillating for method contains() difference was ± 12.4 ns/op).

Second finding is, that hypothesis of theoretical complexities of both HashSet and TreeSet for get operations are (close to being) correct. Therefore it is possible to accept this hypothesis.

7.3.5 Remove Operation – Operation remove() in Set Interface

The remove operation uses remove() method based on Collection interface.

To remove elements from the specific set implementation I have created method removeRandomInSet. This method (Listing 7.10) iterates over array of generic objects and removes every object in the Set implementation.

Listing 7.10: Method removeRandomInSet removes elements from Set implementation

```
<T> void removeRandomInSet(Set<T> filledList, T[] objects,
                    int[] order, int noObjects, Blackhole blackhole) {
    for (int i = 0; i < noObjects; i++) {
        blackhole.consume(filledList.remove(objects[order[i]]));
    }
}
```

This operation is supposed to be $\mathcal{O}(1)$ in case of good hash function and $\mathcal{O}(n)$ for constant hash in HashSet and $\mathcal{O}(log(n))$ for TreeSet.

7.3.5.a Operation remove() in Set Interface – Graphic Results

Figure 7.27: Java 7 Operation remove() on Set

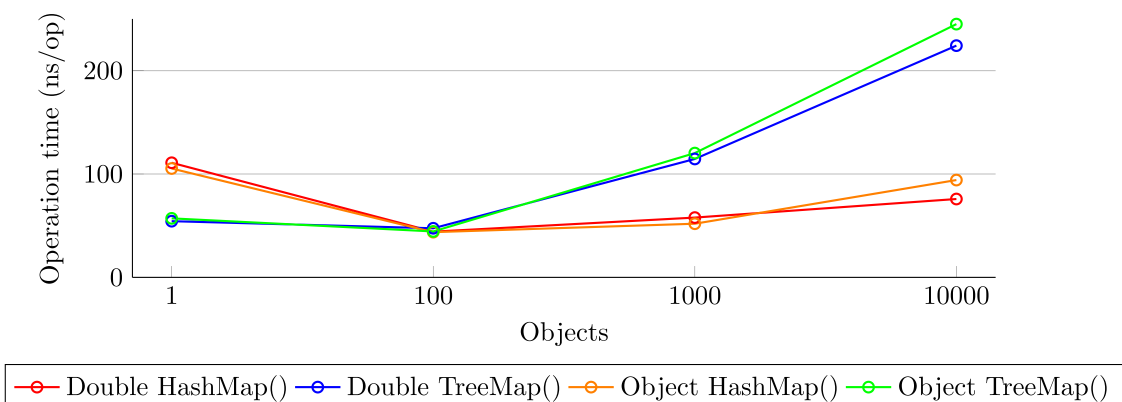


Figure 7.28: Java 8 Operation remove() on Set

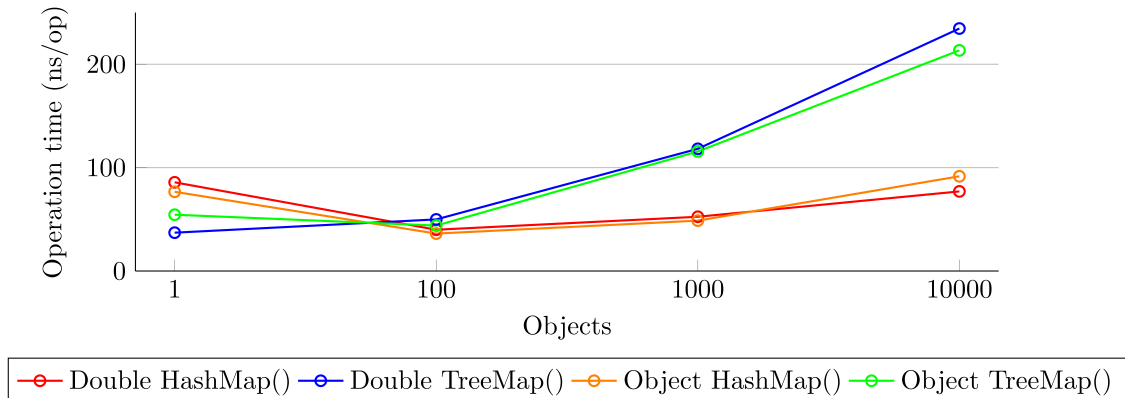


Table 7.20: Average time for remove(Object) operation in HashSet (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – HashSet	108.09	43.99	54.87	84.98
Java 8 – HashSet	81.21	38.06	50.60	84.31
Difference	-26.88	-5.93	-4.28	-0.67

Table 7.21: Average time for remove(Object) operation in TreeSet (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – TreeSet	54.36	45.87	86.20	231.85
Java 8 – TreeSet	45.72	48.70	116.37	229.36
Difference	-8.64	2.84	30.18	-2.50

7.3.6 Operation Remove on Sets in Java Collections Framework – Findings

First interesting finding from the graphs is that removing objects with multiple variables is on average as fast as removing auto-boxed primitive types. It has basically no difference (time difference between removing Object and removing Double is oscillating for each case differently for method remove() difference was $\pm 14.2\ ns/op$).

Second finding is, that hypothesis of theoretical complexities of both HashSet and TreeSet for remove operations are (close to being) correct. Therefore it is possible to accept this hypothesis.

7.4 Java Collections Framework – Map Interface Performance

Map interface in Java Collections Framework has multiple implementations, but the most often used Map implementations are HashMap and TreeMap. These two implementations were tested and compared in this section.

HashMap

Insert operation best/worst case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Get operation case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Remove operation case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Space complexity – $\mathcal{O}(n)$

TreeMap

Insert operation case performance – $\mathcal{O}(log(n))$
Get operation case performance – $\mathcal{O}(log(n))$
Remove operation case performance – $\mathcal{O}(log(n))$
Space complexity – $\mathcal{O}(n)$

7.4.1 Insert Operation – Operation put() in Map Interface

The insert operation uses put() method based on Map interface.

To insert all elements inside the specific map implementation I have created method fillMap. This method (Listing 7.11) iterates over two arrays of generic objects and adds every specified as key and other as value in the Map implementation.

Listing 7.11: Method fillMap inserts elements into Map implementation

```
<K, V> void fillMap(Map<K, V> mapToFill, K[] fillingArrayKeys,
                              V[] fillingArrayValues, int noObjects) {
    for (int i = 0; i < noObjects; i++) {
        mapToFill.put(fillingArrayKeys[i], fillingArrayValues[i]);
    }
}
```

This operation is supposed to be $\mathcal{O}(1)$ in case of good hash function and $\mathcal{O}(n)$ for constant hash in HashMap and $\mathcal{O}(log(n))$ for TreeMap.

7.4.1.a Operation put() in Map Interface – Graphic Results

Figure 7.29: Java 7 Operation put() on Map – Double, String

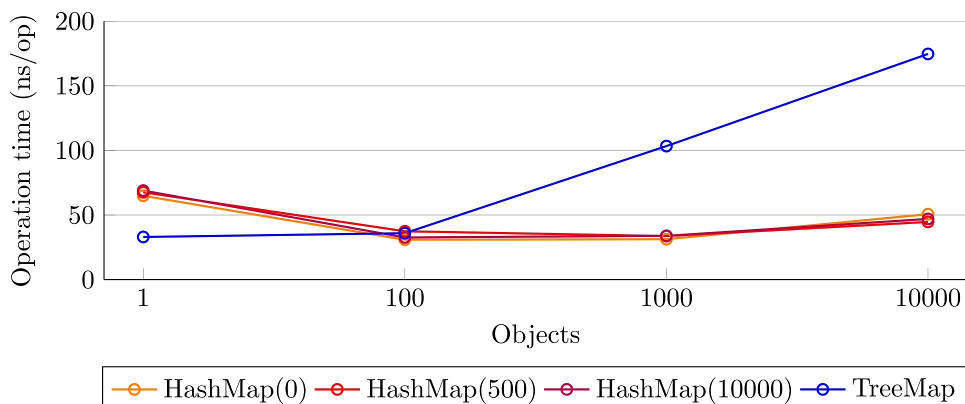


Figure 7.30: Java 8 Operation put() on Map – Double, String

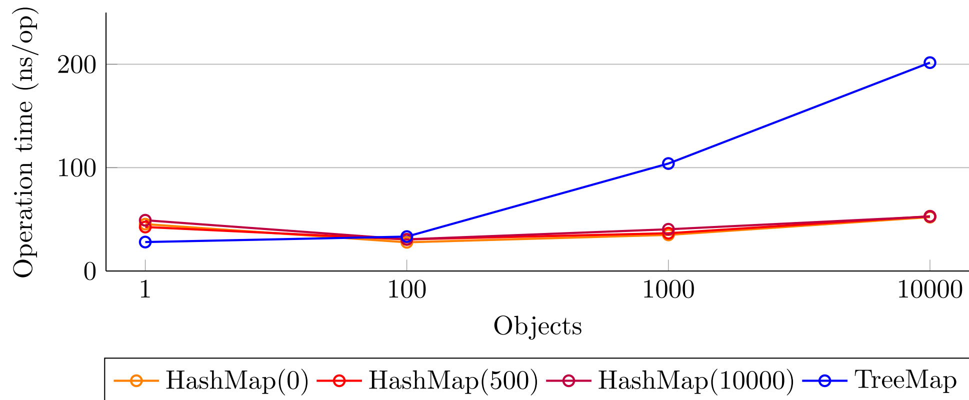


Figure 7.31: Java 7 Operation put() on Map – MockObject, Double

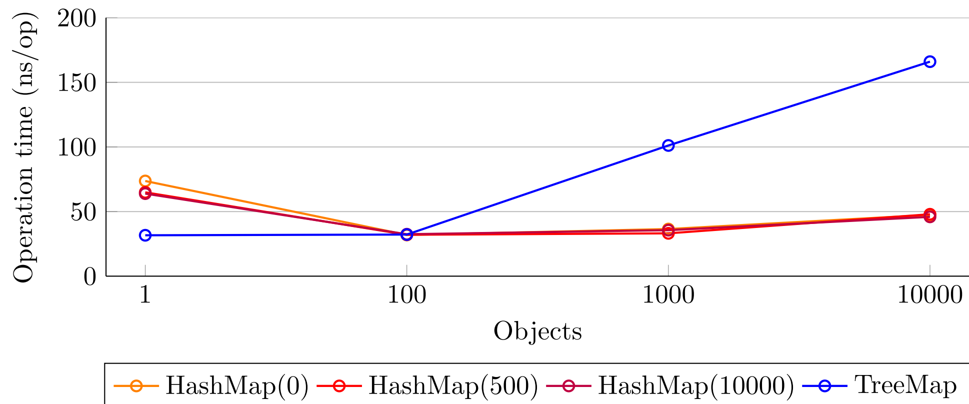


Figure 7.32: Java 8 Operation put() on Map – MockObject, Double

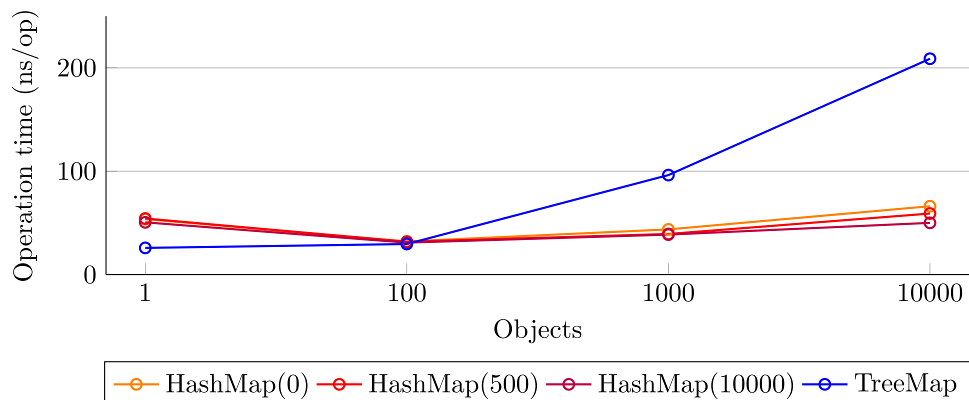


Table 7.22: Average time for put() operation in HashMap (ns/op)

	1	100	1.000	10.000
Java 7 – HashMap	60.14	31.87	37.87	52.75
Java 8 – HashMap	59.98	32.63	36.77	52.93
Difference	-0.16	0.76	-1.05	0.18

Table 7.23: Average time for put() operation in TreeMap (ns/op)

	1	100	1.000	10.000
Java 7 – TreeMap	28.74	30.90	98.71	187.39
Java 8 – TreeMap	29.42	32.76	99.81	191.72
Difference	0.68	1.86	-1.10	4.33

7.4.2 Operation Insert on Maps in Java Collections Framework – Findings

First finding from the graphs is that adding objects with multiple variables as keys with some value is as fast as adding auto-boxed primitive types as keys with some value. It has basically no difference (time difference between adding as key Object in contrary with Double is oscillating for method put() difference was $\pm 0.7\ ns/op$).

Second finding is, that the HashMap has roughly the same time complexity for pre-set capacity parameter (based on the three sizes from experiment benchmarking – 0, 500 and 10000). The biggest difference is in adding only one element where the difference is in time spent pre-allocating the buffer with the specified size.

Third finding is, that hypothesis of theoretical complexities of both HashMap and TreeMap for insert operations are (close to being) correct. Therefore it is possible to accept this hypothesis.

7.4.3 Get Operation – Operation get() in Map Interface

The get operation uses get() method based on Map interface.

To get elements from the specific map implementation I have created method getRandomFromMap. This method (Listing 7.12) iterates over array of generic objects and gets every object in the Map implementation.

Listing 7.12: Method fillList getRandomFromMap fetches elements from Map implementation

```
<K, V> void getRandomFromMap(Map<K, V> filledMap, K[] keys,
                    int[] order, int noObjects, Blackhole blackhole) {
    for (int i = 0; i < noObjects; i++) {
        blackhole.consume(filledMap.get(keys[order[i]]));
    }
}
```

This operation is supposed to be $\mathcal{O}(1)$ in case of good hash function and $\mathcal{O}(n)$ for constant hash for HashSet and $\mathcal{O}(log(n))$ for TreeSet.

7.4.3.a Operation get() in Map Interface – Graphic Results

Figure 7.33: Java 7 Operation get() on Map

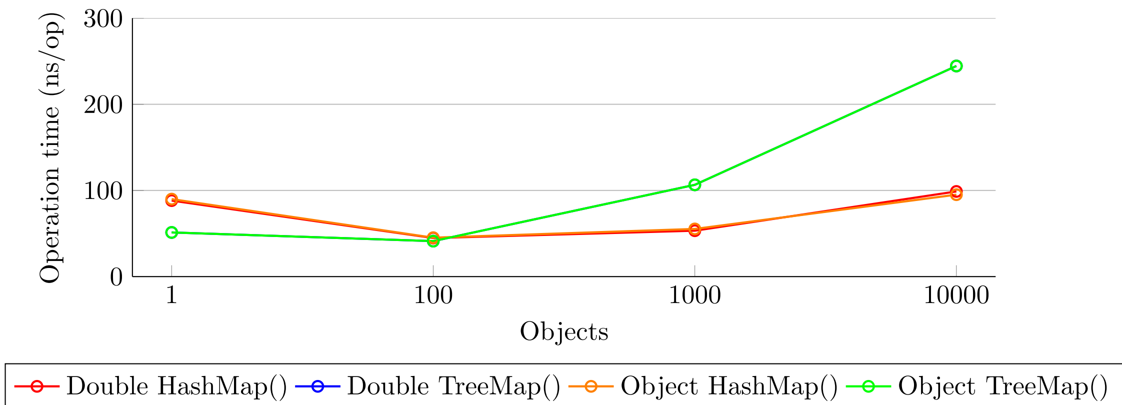


Figure 7.34: Java 8 Operation get() on Map

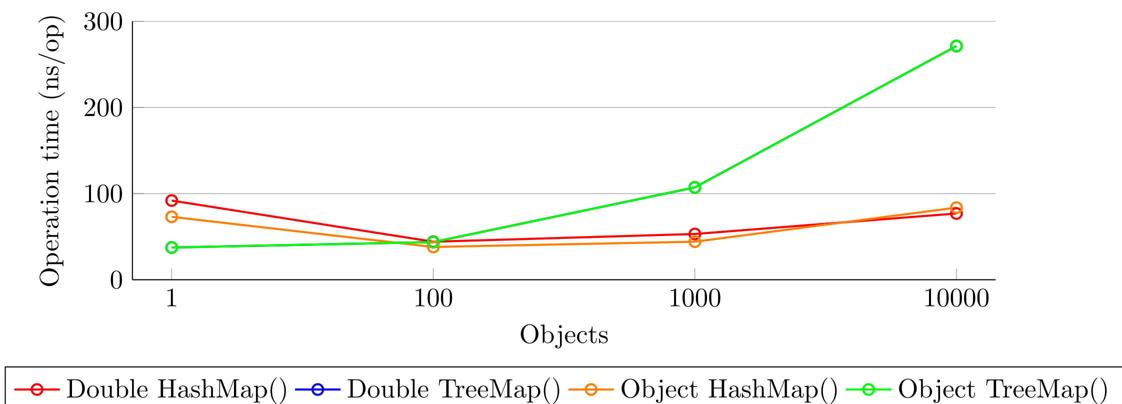


Table 7.24: Average time for get() operation in HashMap (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – HashMap	89.09	44.97	54.30	96.97
Java 8 – HashMap	82.49	41.11	48.66	80.31
Difference	-6.60	-5.64	-5.64	-16.66

Table 7.25: Average time for get() operation in TreeMap (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – TreeMap	42.21	38.82	103.55	239.53
Java 8 – TreeMap	35.50	42.87	108.28	236.74
Difference	-6.71	4.04	4.73	-3.79

7.4.4 Get Operation – Operation contains() in Map Interface

The get operation uses contains() method based on Map interface.

To get elements from the specific map implementation I have created method containsInSet. This method (Listing 7.13) iterates over array of generic objects and tries object keys containing in the Map implementation.

Listing 7.13: Method containsInMap elements into List implementation

```
<K, V> void containsInMap(Map<K, V> filledMap, K[] keys,
                    int[] order, int noObjects, Blackhole blackhole) {
    for (int i = 0; i < noObjects; i++) {
        blackhole.consume(filledMap.containsKey(keys[order[i]]));
    }
}
```

This operation is supposed to be $\mathcal{O}(1)$ in case of good hash function and $\mathcal{O}(n)$ for constant hash for HashSet and $\mathcal{O}(log(n))$ for TreeSet.

7.4.4.a Operation contains() in Map Interface – Graphic Results

Figure 7.35: Java 7 Operation contains() on Map

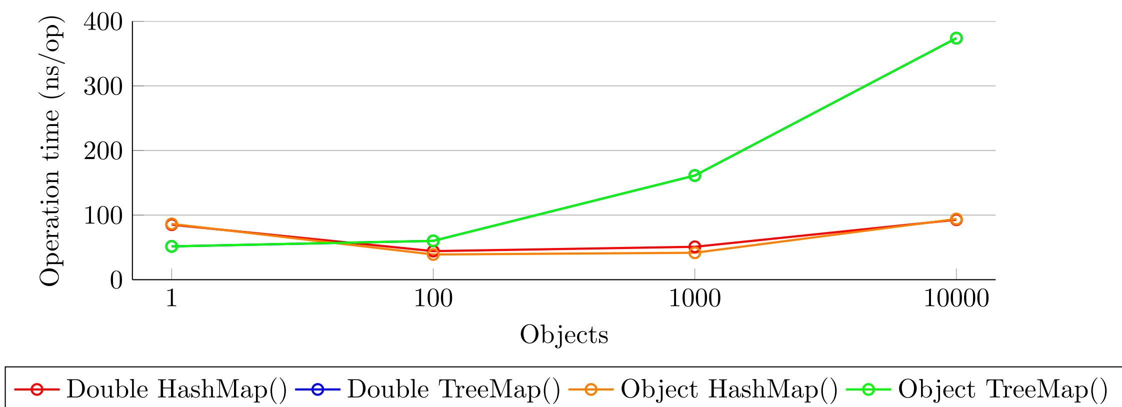


Figure 7.36: Java 8 Operation contains() on Map

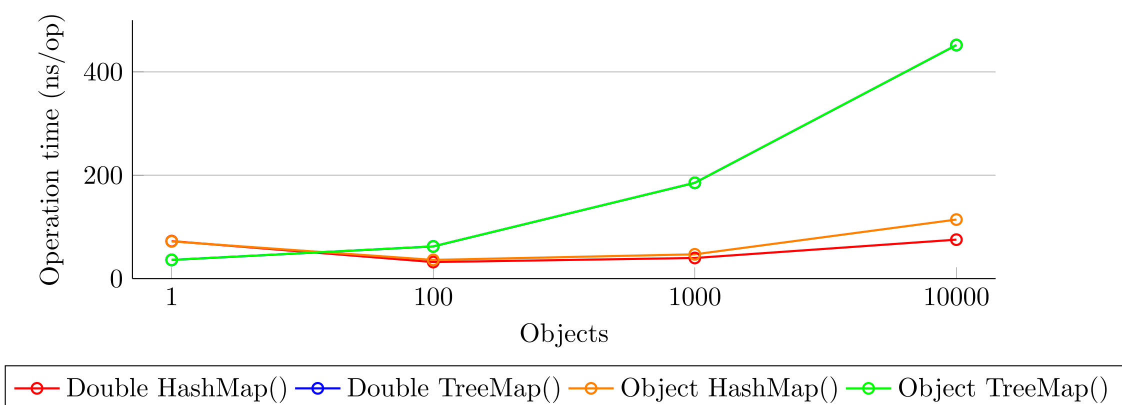


Table 7.26: Average time for contains() operation in HashMap (ns/op)

	1	100	1.000	10.000
Java 7 – HashMap	85.76	41.63	46.31	93.38
Java 8 – HashMap	72.26	34.14	43.48	94.83
Difference	-13.50	-7.49	-2.83	1.44

Table 7.27: Average time for contains() operation in TreeMap (ns/op)

	1	100	1.000	10.000
Java 7 – TreeMap	52.86	61.59	165.24	383.15
Java 8 – TreeMap	37.08	53.84	189.97	463.11
Difference	-15.78	2.25	24.72	79.96

7.4.5 Operation Get on Maps in Java Collections Framework – Findings

First interesting finding and quite obvious from the graphs is that getting elements as keys of type object with multiple variables is as fast as getting auto-boxed primitive types. The difference is minor and can be taken as unimportant factor (average difference was $\pm 12.31\ ns/op$).

Second finding is, that hypothesis of theoretical complexities of both HashMap and TreeMap for get operations are (close to being) correct. Therefore it is possible to accept this hypothesis.

7.4.6 Remove operation – Operation remove() in Map Interface

The remove operation uses remove() method based on Collection interface.

To remove elements from the specific map implementation I have created method removeRandomInMap. This method (Listing 7.14) iterates over array of generic objects and removes objects from the Map implementation.

Listing 7.14: Method removeRandomInMap removes elements from List implementation

```
<K, V> void removeRandomInMap(Map<K, V> filledMap, K[] keys,
                    int[] order, int noObjects, Blackhole blackhole) {
    for (int i = 0; i < noObjects; i++) {
        blackhole.consume(filledMap.remove(keys[order[i]]));
    }
}
```

This operation is supposed to be $\mathcal{O}(1)$ in case of good hash function and $\mathcal{O}(n)$ for constant hash for HashSet and $\mathcal{O}(log(n))$ for TreeSet.

7.4.6.a Operation remove() in Map Interface – Graphic Results

Figure 7.37: Java 7 Operation remove() on Map

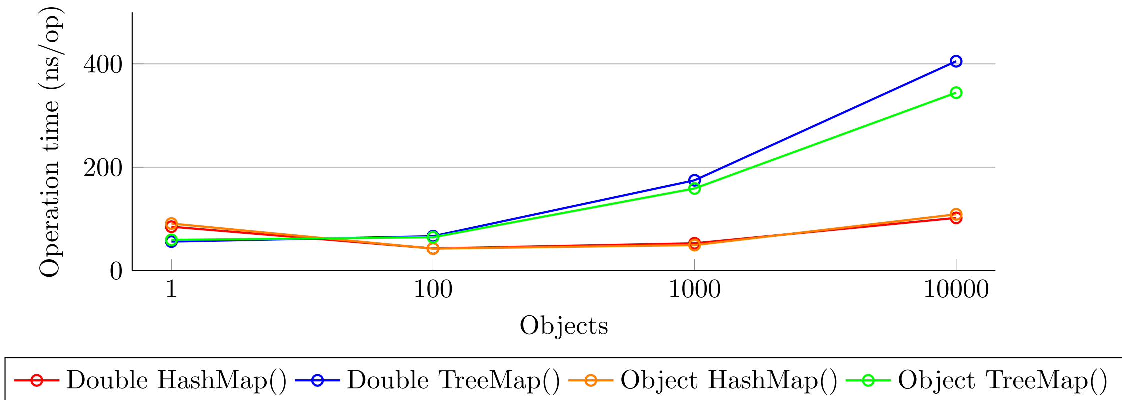


Figure 7.38: Java 8 Operation remove() on Map

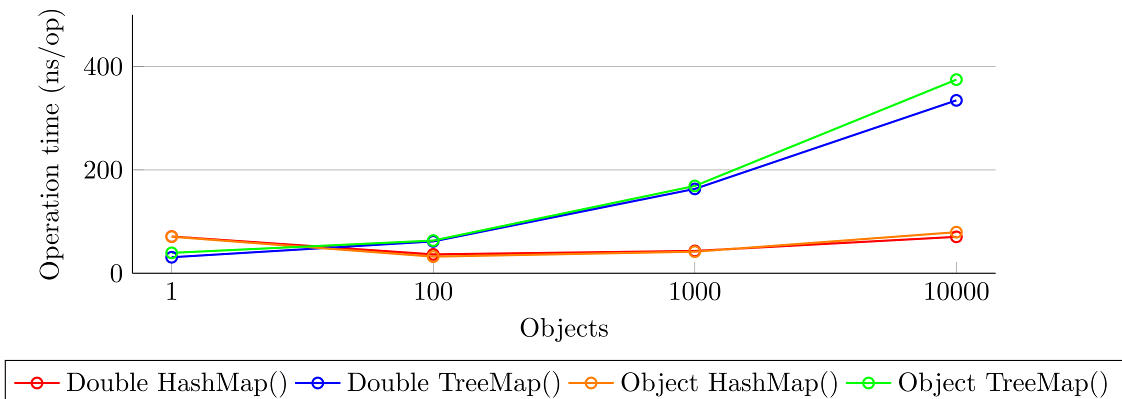


Table 7.28: Average time for remove(Object) operation in HashMap (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – HashMap	103.27	46.45	60.75	95.71
Java 8 – HashMap	100.41	37.86	51.74	76.87
Difference	-2.86	-8.59	-9.01	-18.84

Table 7.29: Average time for remove(Object) operation in TreeMap (ns/op)

	1	**100**	**1.000**	**10.000**
Java 7 – TreeMap	37.18	49.45	198.49	389.91
Java 8 – TreeMap	36.49	49.42	193.69	373.45
Difference	-0.68	-0.04	-4.80	-16.46

7.4.7 Operation Remove on Map in Java Collections Framework – Findings

First interesting finding and quite obvious from the graphs is that removing objects with multiple variables is as fast as removing auto-boxed primitive types. The difference is minor and can be taken as unimportant factor (average difference was ± 12.21 ns/op).

Second finding is, that hypothesis of theoretical complexities of both HashMap and TreeMap for add operations are (close to being) correct. Therefore it is possible to accept this hypothesis.

7.5 Java Collections Framework – Results anomaly

The anomaly has happened while testing. Some test results are showing better performance for Object with multiple auto-boxed data types while the auto-boxed data type, Double, has performance results less impressive.

Listing 7.15: Implementation of equals() method in Double

```
public boolean equals(Object obj) {
    return (obj instanceof Double)
           && (doubleToLongBits(((Double)obj).value) ==
                  doubleToLongBits(value));
}
```

Listing 7.16: Implementation of doubleToLongBits(double) method in Double

```
public static long doubleToLongBits(double value) {
    long result = doubleToRawLongBits(value);
    // Check for NaN based on values of bit fields, maximum
    // exponent and nonzero significand.
    if ( ((result & DoubleConsts.EXP_BIT_MASK) ==
          DoubleConsts.EXP_BIT_MASK) &&
          (result & DoubleConsts.SIGNIF_BIT_MASK) != 0L)
          result = 0x7ff8000000000000L;
    return result;
}
```

It is probably result of the more complex implementation of equals(object) method in Double than in Object equals().

Listing 7.17: Implementation of equals() method in Object

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

7.6 Java Collections Framework – Parallel Collections

I have tried to do the tests for multiple threads for add operation with classes from java.utils.concurrent and by encapsulation via Collection.synchronizedList.

Results for CopyOnWriteArrayList and encapsulated ArrayList in synchronized context from Collection were not as expected. Results shown increased time inefficiency with increased number of threads.

Figure 7.39: Operation add() on parallel ArrayLists – Double

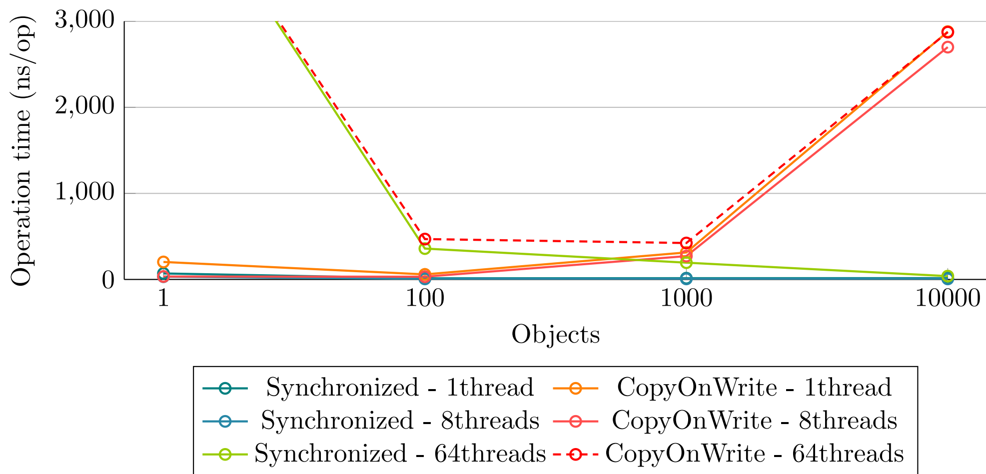
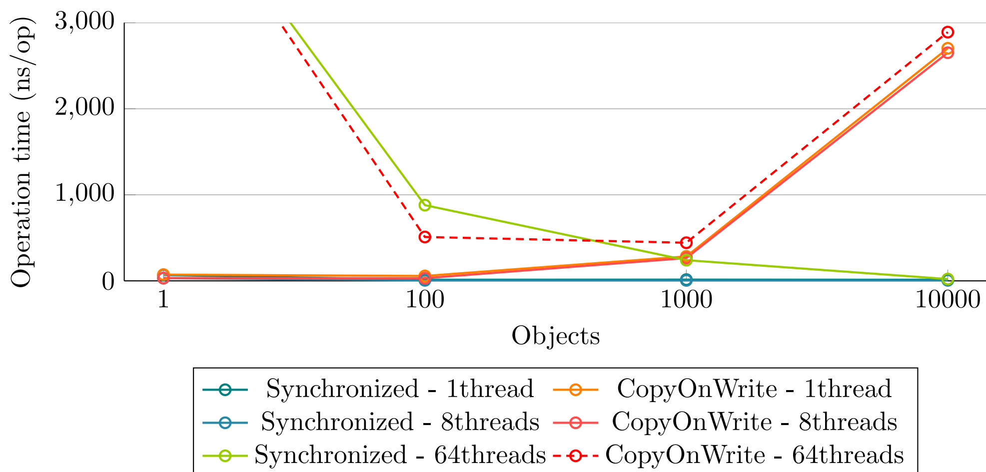


Figure 7.40: Operation add() on parallel ArrayLists – MockObject



The results were counted for method add() call in Lists interface. Graphs show time spent divided by number of elements times number of threads.

Proper method handling for multiple tests will require bigger effort. The results may be influenced by processor type and number of cores and hyper-threading.

8 Java External Collections Frameworks

This chapter provides graphic comparison of Collections Frameworks (Section 4.1, page 23). The Collection frameworks shown in this chapter were selected because they add enhancements to basic implementations. Graphs show differences in contrast with Java Collections Framework implementations.

8.1 List Interface Variants Performance

On foundations of Java Collections Framework – List Interface Performance (Section 7.2, page 33) I will show the differences of the external collections frameworks.

List interface in Java Collections Framework has multiple implementations, but the most often used List implementations are ArrayList and LinkedList.

ArrayList, which is base on an array, has complexity:
Insert operation case performance – amortized $\mathcal{O}(1)$
Get operation case performance – $\mathcal{O}(1)$
Remove operation case performance – $\mathcal{O}(n)$
Space complexity – $\mathcal{O}(n)$

LinkedList, which is base on linked nodes, has time complexity:
Insert operation best/worst case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Get operation case performance – $\mathcal{O}(n)$
Remove operation best/worst case performance – $\mathcal{O}(1)/\mathcal{O}(n)$
Space complexity – $\mathcal{O}(n)$

8.1.1 Insertion Operation in List Structure

On foundations of Insert Operation in List Interface (Section 7.2.1, page 34) I will show the differences of the external collections frameworks.

ArrayLists complexity is amortized $\mathcal{O}(1)$ (2.5.3). The amortization is important, because of occasional increase of array size, which is $\mathcal{O}(n)$ that is needed to move all n elements to the newly created bigger array.

LinkedLists complexity is somewhere between 1 and $n/2$. The basic addition of element after last node in LinkedList or before first node is 1 as both sides of the LinkedList are the starting nodes. Adding the element in the middle has a worst case scenario, that is $n/2$, because it is needed to iterate over nodes to get in the place. It is 'only' $n/2$, because of two possible starting points at the beginning or the end of the LinkedList. If the index is closer to the end iteration starts with last node, else iteration starts with first node.

8.1.2 Insertion Operation – Operation add() in List Structure

To insert all elements inside the specific list implementation I have created and overloaded method fillList. These methods iterate over array of primitive data types and adds every primitive data type in the List implementation.

This operation is supposed to be amortized $\mathcal{O}(1)$ for ArrayList and $\mathcal{O}(1)$ for LinkedList.

8.1.2.a Operation add() in List Structure – Graphic Results

Figure 8.1: HPPC DoubleIndexedContainer Operation Add() on Lists – double

Figure 8.2: Trove TDoubleList Operation add() on Lists – double

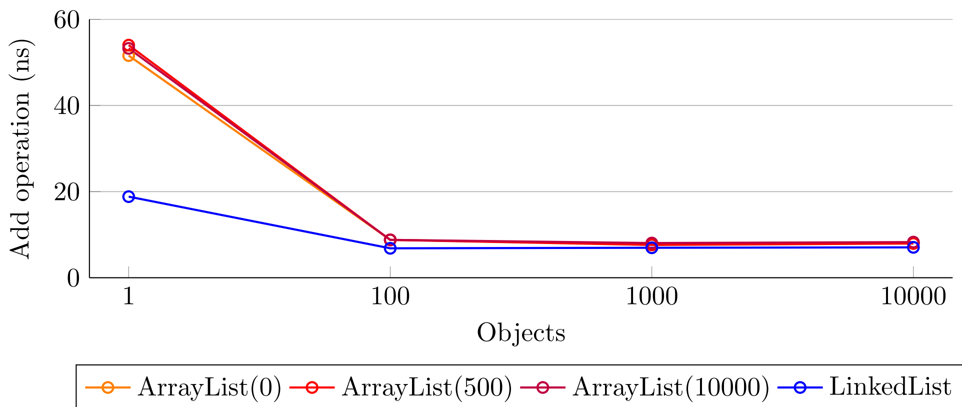
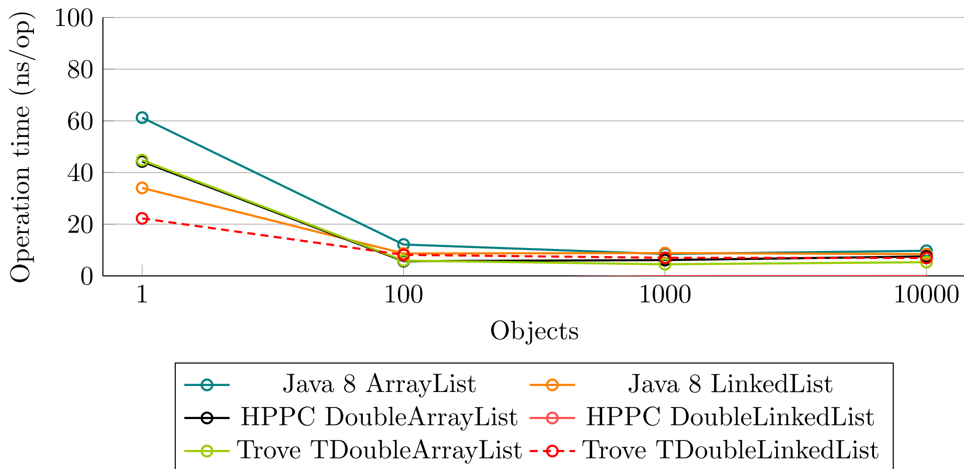


Figure 8.3: Comparison of Operation Add() on List implementations – Double / double



8.1.2.b Operation add() in List Structure – Findings

First finding, that all implementations of ArrayList have roughly the same time complexity for pre-set capacity parameter (based on the three sizes from experiment benchmarking – 0, 500 and 10000). The biggest difference is in adding only one element which is based on creating the list.

Second finding is, that in all implementations have comparable time complexity for add() method.

8.1.3 Get Operation in List Structure

On foundations of Get Operation in List Interface (Section 7.2.6, page 41) I will show the differences of the external collections frameworks.

ArrayLists complexity of method get() is $\mathcal{O}(1)$.

LinkedLists complexity of method get() is $n/2$ with best case equal $\mathcal{O}(1)$.

8.1.4 Get Operation – Operation get(index) in List Structure

To fetch elements from the list implementation I have created and overloaded method getRandomFromList. These methods iterate over list and randomly fetches objects.

This operation is supposed to be equal $\mathcal{O}(1)$ for ArrayList and $n/2$ for LinkedList.

ArrayLists complexity of method get() is $\mathcal{O}(1)$.

LinkedLists complexity of method get() is $n/2$ with best case equal $\mathcal{O}(1)$.

8.1.4.a Operation get() in List Structure – Graphic Results

Figure 8.4: Comparison of Operation get() on Lists – Double / double

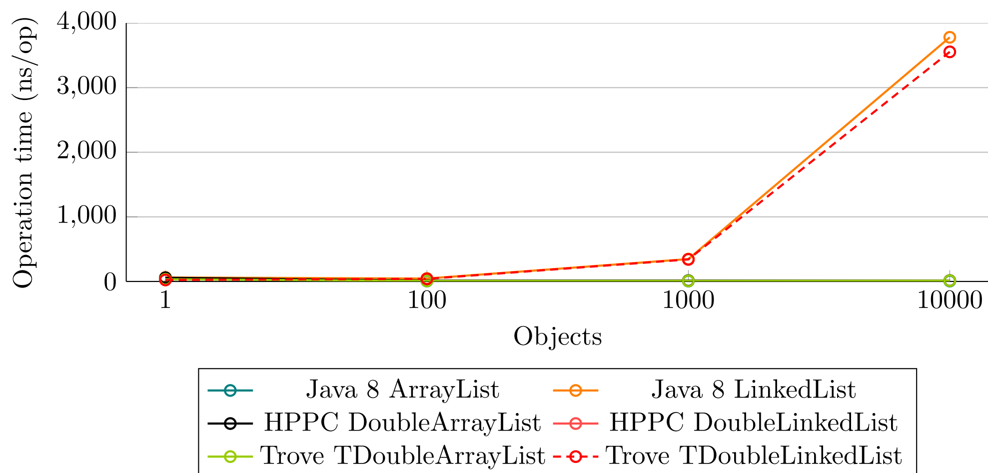
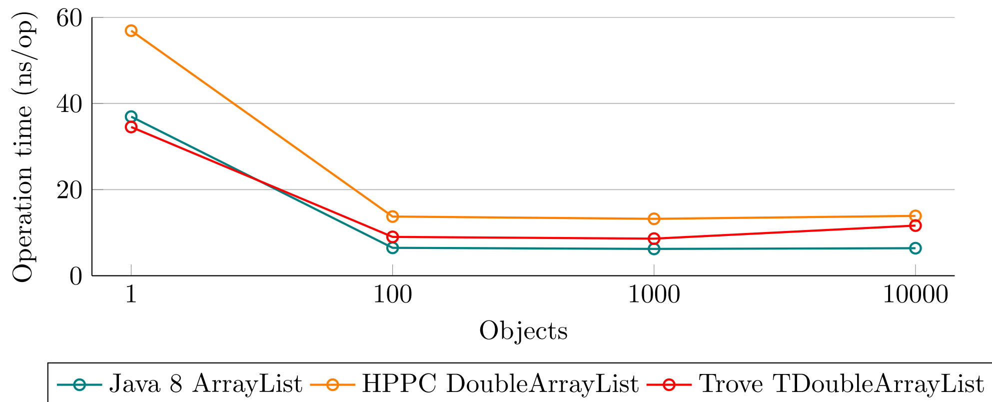


Figure 8.5: Comparison of Operation get() on ArrayLists – Double / double



8.1.4.b Operation get() in List Structure – Findings

First finding is, that ArrayList implementations for primitive data types are slower than basic Java 8 implementation of ArrayList for get() method. Their priority is set on memory efficiency, not on time complexity.

Second finding is, that Trove LinkedList implementation for primitive data types is faster than basic Java 8 implementation of LinkedList for get() method.

9 Conclusion

In this bachelor thesis I have described basic classes and structures used in collection frameworks. Main framework was Java Collections Framework provided by Java Virtual Machine, but several open source frameworks and libraries were included for comparison. These frameworks and libraries are widely used and it was interesting to test them.

While creating this paper I gained hands on experience and understanding of how collections theoretically work. I have learned how to create reliable microbenchmarks and graphs according. With those newly earned experience I have compared the frameworks, created visualisations of the comparison results and collated them into this work.

After mutual agreement with my advisor we decided to concentrate on Collections and omit the Caches.

With created comparisons I have proven that the the Java Collections Framework is as efficient as described in official documentation. Graphs shown findings like that ArrayLists implementation of dynamic array is very effective and so its performance was not determined by initial capacity. Graphs have shown that some implementations of external frameworks display slight improvements in time complexity. Further more primitive data type collections provide better memory efficiency as they omit the auto-boxing encapsulation of primitive data types.

When focusing on Collections I have tackled multiple setbacks. One of the problems was inconsistency of testing results. While results from Java 8 on Windows shown improvements of performance the results for Ubuntu with Java 7 were very highly optimized and so tests often looked like Java 8 have not only stagnated but even decreased it's performance. Different kind of setback was HPPC esoteric library which added the classes that were excluded from basic HPPC library. I had to omit this library as compilation was always failing.

The biggest advantage of frameworks was not the improvements of performance. It was the addition of functionality not provided in original collections or utility classes.

9.1 Personal Influence

Studying the subject of this bachelor thesis enhanced further experience with efficiency of algorithms and behaviour of the classes.

While testing some collection classes I understood behaviour behind their structure. For example realizing that multiple set structures are created by having the corresponding map collection. Which means that instead of object structure there is key, value structure. On the other side EnumSet with just value of long was pleasant surprise.

9.2 Follow-up Work

While hardware is still improving the software will have to adapt. With increasing number of cores in processors programmers will be forced to use multi-threading processing more often.

As written in abstract "This work should be base for the future investigation of tests in parallel data processing, maybe distributed frameworks". The follow-up research will be aimed at parallel environment and Lambda expressions.

I am enthusiastic about future efficiency of data processing. The bachelor thesis gave me insight and desire to work on further development of this topic.

10 Bibliography

[1] D. E. Knuth, “Big omicron and big omega and big theta,” *ACM Sigact News*, vol. 8, no. 2, pp. 18–24, 1976.

[2] T. H. Cormen, *Introduction to algorithms.* MIT press, 2009.

[3] “Java platform, standard edition (java se) 8.” `https://docs.oracle.com/javase/8/`, 2014.

[4] “Collections framework enhancements in java se 8.” `http://docs.oracle.com/javase/8/docs/technotes/guides/collections/changes8.html`, 2014.

[5] “Collections framework enhancements in java se 7.” `http://docs.oracle.com/javase/8/docs/technotes/guides/collections/changes7.html`, 2011.

[6] M. Fowler, *Patterns of enterprise application architecture.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[7] D. Lea, J. Bowbeer, D. Holmes, and S. AG, “Jsr 166: Concurrency utilities,” *URL http://jcp. org/en/jsr/detail*, 2004.

[8] B. Goetz *et al.*, “Jsr 335: Lambda expressions for the javatm programming language,” 2014.

[9] M. A. Weiss and S. Hartman, *Data structures and problem solving using Java*, vol. 204. Addison-Wesley Reading, 1998.

[10] J. Zukowski, *Java collections.* Apress, 2001.

[11] M. Naftalin and P. Wadler, *Java generics and collections.* “ O’Reilly Media, Inc.”, 2006.

[12] C. Horstmann and G. Cornell, “Core java fundamentals (vol i and ii),” 2001.

[13] M. A. Weiss, *Data Structures and Algorithm Analysis in Java. Mark Allen Weiss.* Pearson education, 2012.

[14] A. Drozdek, *Data Structures and algorithms in C++.* Cengage Learning, 2012.

[15] C. Hunt and B. John, *Java performance.* Prentice Hall Press, 2011.

[16] J. Shirazi, *Java performance tuning.* “ O’Reilly Media, Inc.”, 2003.

[17] O. Oransa, *Java EE 7 Performance Tuning and Optimization.* Packt Publishing Ltd, 2014.

[18] R. Pecinovský, *OOP a Java 8: Návrh a vývoj složitějšího projektu vyhovujícího zadanému rámci*, vol. 1. Tomáš Bruckner, 2015.

[19] J. Juneau, *Java 8 Recipes.* Apress, 2014.