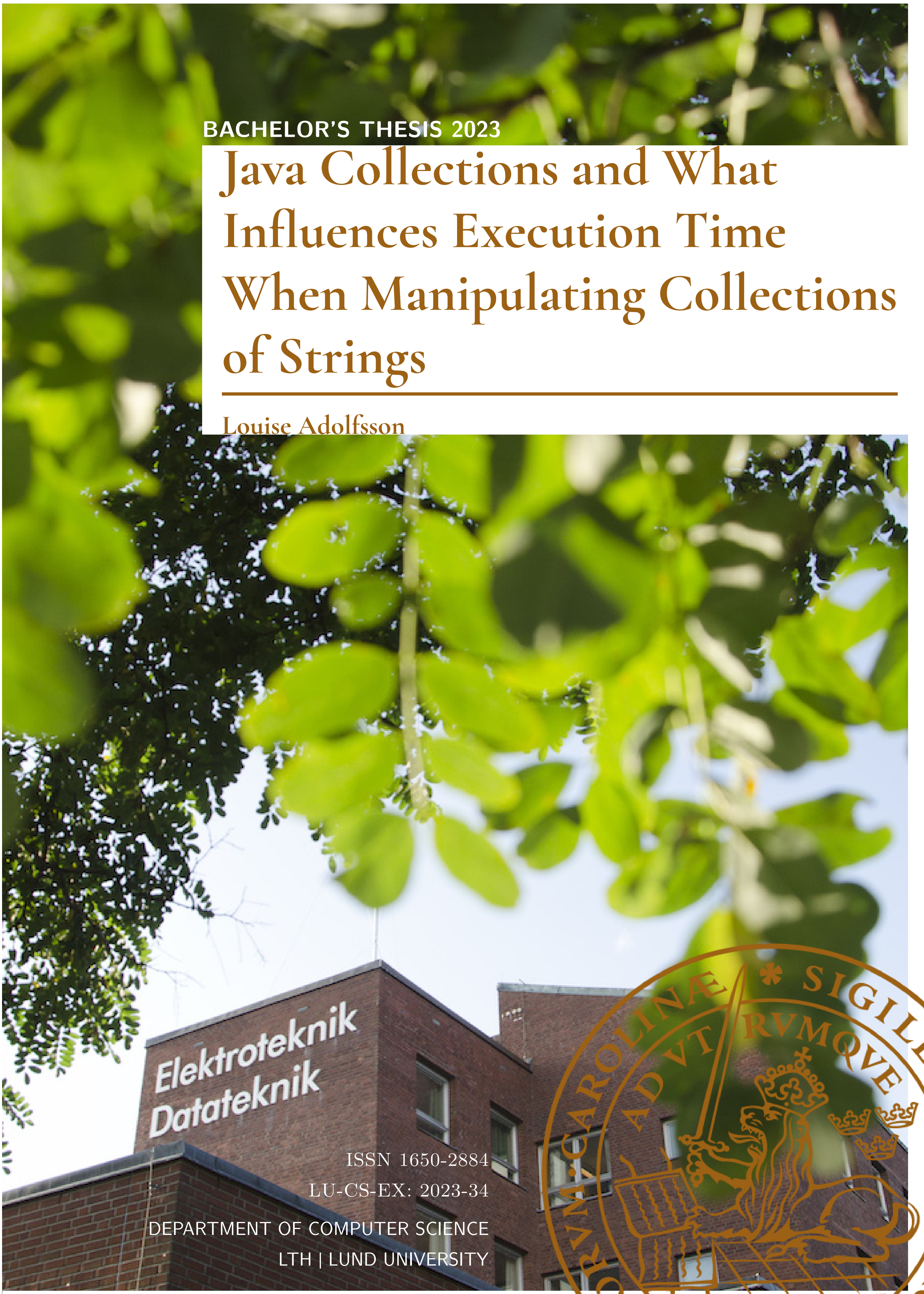


BACHELOR'S THESIS 2023

Java Collections and What Influences Execution Time When Manipulating Collections of Strings

Louise Adolfsson

ISSN 1650-2884
LU-CS-EX: 2023-34

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

KANDIDATARBETE
Datavetenskap

LU-CS-EX: 2023-34

Java Collections and What Influences Execution Time When Manipulating Collections of Strings

Samlingar av Java strängar och vad som påverkar utförandetiden när man hanterar samlingar av strängar

Louise Adolfsson

Java Collections and What Influences Execution Time When Manipulating Collections of Strings

(A study using benchmarks)

Louise Adolfsson
`lo6806ad-s@cs.lth.se`

August 16, 2023

Bachelor's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Noric Couderc, `noric.couderc.cs.lth.se`

Examiner: Görel Hedin, `gorel.hedin@cs.lth.se`

Abstract

While manipulating collections of strings in Java there are multiple data structures to use and there can be multiple things affecting the execution time, for example, the size of the strings, how big the collection is and which collection implementation is being used.

The existing literature on collection implementations focus on the size of the collection for predicting the cost of operations. For map implementations other factors can be important, such as the cost of hashing keys, comparing them, or the cost of resizing the map as more elements are inserted. What is the influence of these factors on the cost of map operations? Is the current documentation correct? This paper will try to answer these questions using benchmarks.

The experiments shows that for the TreeMap the map size is the only thing affecting its execution time. For the OpenHashMap, LinkedHashMap and HashMap the loadfactor, starting capacity and the collision rate can affect the execution time for the maps. The documentation for the maps is generally correct, however when doing many insertions in the HashMap at once the execution time is reduced when the loadfactor is increased and when the starting capacity is reduced. This behavior is contradicting to how the documentation describes it.

Keywords: Java collections, influencing execution time, hash map, tree map, linked hash map, open hash map

Acknowledgements

I want to thank Noric Couderc who have been my supervisor throughout this project, his guidance and support is what made this possible.

Contents

Chapter 1

Introduction

When manipulating collections of strings, knowing what factors can possibly affect the execution time and how those factors affect it, is important if one wants to maximize efficiency. In the literature today it is generally only discussed how the size of the collection affects time performance.

The books *Introduction to Algorithms*[1] and *The Algorithm Design Manual[6]* describe how some specific collections operations are dependent of the size of their collection. In *Introduction to Algorithms* the authors describe hash tables, binary search trees and Red-Black trees. The Java HashMap is an implementation of the hash table and is described in the book as its operations having a complexity of $O(1)$ in a best case scenario with very few collisions. In the worst case the complexity is $O(n)$, with n being the collection size, when there are many collisions. The Java TreeMap is an implementation of a Red-Black tree and it is described in the book as having a complexity of $O(\log(n))$, where n is the size of the collection, for its simple operations. *The Algorithm Design Manual* describes the complexities for the hash table and Red-Black tree in the same way, but neither books talk about other possible influences on the performance of the collections.

1.1 Purpose

This paper have two research questions that will be studied with experimentation

- What affects the execution time for simple operations on maps other than the size of the map?
- Is the current documentation of the maps correct?

1.2 Constraints

The compiler version, garbage collection or computer architecture can also affect execution time, but these will not be taken into account in this paper. Memory usage can also be an important aspect when working with collections but in this paper, I will focus on execution time only.

In this paper I will look at the java HashMap, LinkedHashMap, TreeMap and the Object2ObjectOpenHashMap from the fastutil java library, I will refer to this as OpenHashMap.

1.3 Outline

In chapter 2, I describe the methods of how the string collections have been benchmarked and analyzed. Chapter 3 presents the results of the benchmarks and analysis. Chapter 4 consists of the discussion of the results. Chapter 5 describes the threats to validity. Chapter 6 consists of the future work. Chapter 7 contains the conclusions of the results of this paper.

1.4 Terminology

There are some important terms that are used often through out this paper and are needed to understand most of the content.

Simple operations: All the maps have a get and a put operation, I will refer to these two operations as the maps simple operations through out the paper.

Rehash: Rehashes is when the map resize to double its capacity and its internal data structures are rebuilt. A example of what a rehash could look like can be seen in figure 1.1

Loadfactor: Loadfactor is a parameter that decides how full a map can be before it will rehash. So if the loadfactor is its default value of 0.75 the map will rehash when it has filled 75% of the map.

Collision: Collision happen when two different strings have the same hash code, as an example the strings "AaAa", "BBBB", "AbBB" and "BBAa" all have the same hash code in java and would in a HashMap and LinkedHashMap cause collision. The Java string hash code is calculated with

$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + ... + s[n-1]$ where n is the length of the string[3].

Slot: Each position in the map is called a slot. In figure 1.1 there are 8 slots before rehash then 16 slots after the rehash.

Capacity: Capacity is the number of slots in the map.

Bucket: A bucket is a slot with a list in that can hold elements that collides. The HashMap and LinkedHashMap have buckets while the TreeMap and OpenHashMap doesn't.

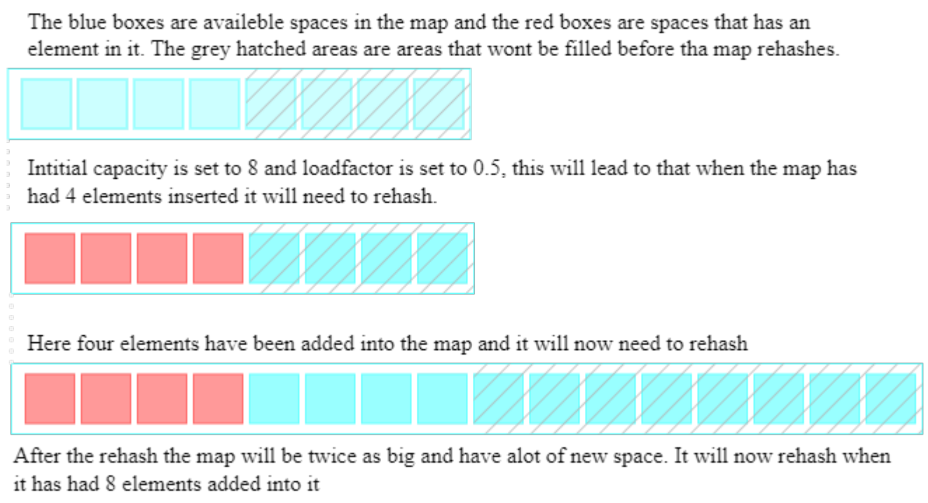

The blue boxes are availeble spaces in the map and the red boxes are spaces that has an element in it. The grey hatched areas are areas that wont be filled before tha map rehashes.
Intitial capacity is set to 8 and loadfactor is set to 0.5, this will lead to that when the map has had 4 elements inserted it will need to rehash.
Here four elements have been added into the map and it will now need to rehash
After the rehash the map will be twice as big and have alot of new space. It will now rehash when it has had 8 elements added into it

Figure 1.1: A figure explaining how loadfactor and starting capacity affects the map size

1.5 Background

Each of the four chosen maps work different and have their own documentation. Here I will describe what we know about the maps based on their documentation.

1.5.1 HashMap and LinkedHashMap

The Java documentation for HashMap[1] and LinkedHashMap[2] describes that both collections can provide a constant time performance for their simple operations given that there aren't to many collisions. The documentations also explain that the initial capacity and loadfactor affect the maps time performance. It states that the loadfactor shouldn't be put too big since that increases the cost of lookups. The documentation also states that one needs to take into account the loadfactor and the expected number of entries into the map when setting the initial capacity to avoid rehashes. Both the maps use a linked list as their buckets implementation, so when collisions happen the strings end up in a liked list.

The LinkedHashMap extends the HashMap. The major difference between the HashMap and the LinkedHashMap is that the LinkedHashMap maintains the order of which keys are inserted into the map while the HashMap does not care about the order. Internally the LinkedHashMap uses a doubly Linked List to keep the order of the keys.

To show what can impact the execution time for simple operations in the hash table and TreeMap I created two causal graphs. These graphs can be seen in figure 1.2 and 1.3. The bottom box in these graphs represents the execution time for simple operations. Each vertex represents variables that the user can set or observe, and when one variable affects another, we connect them by arrows. For example, the size of the hash table affects the number of slots in the table, so we connect them, by arrows, as shown on the top-left corner of the figure 1.2.

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

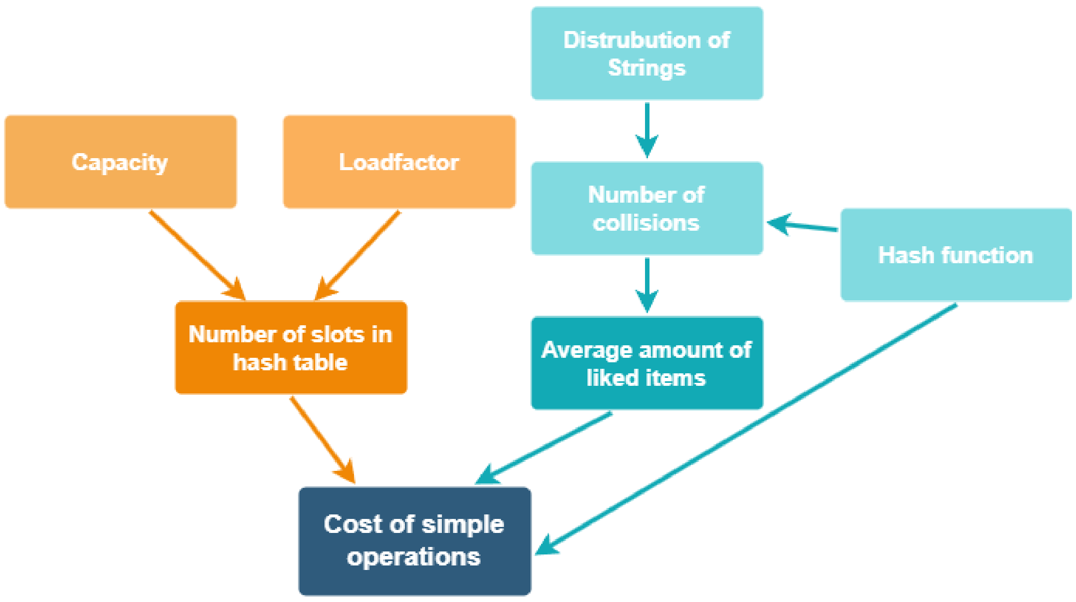

Distrubution of Strings
Capacity
Loadfactor
Number of collisions
Hash function
Number of slots in hash table
Average amount of liked items
Cost of simple operations

Figure 1.2: A Causal graph of what affects the execution time for simple operations on the HashMap and LinkedHashMap. The arrows show what affects what and the boxes are what affects the execution time.

This then points to the cost of simple operations, meaning the number of slots in the hash table affects the cost of simple operations.

Figure 1.2 shows what the hash table's execution time for simple operations is depending on. Starting from the top left in the graph, the capacity. This will affect the number of slots in the table and as such can have an affect on execution time.

Secondly, the loadfactor decides when to increase the number of slots in the map based on the number of entries in the map. The default loadfactor is 0.75, so the map is never more than 75% filled. Having a bigger loadfactor will give less free space in the map, but also fewer rehashes.

Thirdly, the number of collisions will increase the number of average linked items in a map. Having collisions makes the execution time longer since it has to traverse the list in the bucket and do comparisons of each object. In the case of the OpenHashMap that doesn't use a linked list, so it will have not linked items. Instead when there are collisions it will look in the next free slot to put the collided element in. This will also increase the cost of simple operation as the map has to find to the next slot.

Fourth, the distribution of strings, in some cases where collisions happen often, the execution time of some operations can be increased substantially.

Lastly, the execution time is affected by the Hash function. It controls how many collisions there will be and therefore affects the cost of a lookup. But it also directly affects the execution time by how long the function takes. In theory with an expensive hash function it could spend some time just calculating each string's hash code, however since java strings cache the result of the hash function the hash function should only be done once per operation.

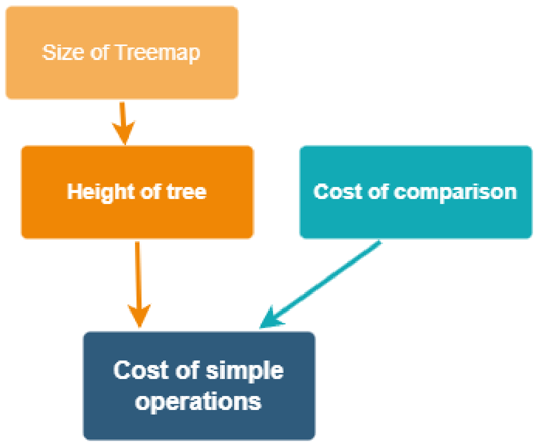


Figure 1.3: A causal graph of what affects the execution time for operations on the TreeMap. The arrows show what affects what and the boxes are what affects the execution time.

1.5.2 TreeMap

The Java documentation for the TreeMap[3] states that the TreeMap provides a *guaranteed* $\log(n)$, where n is the map size, complexity for the simple operations. The documentation does not state any other factors that could affect performance (aside of the cost of comparisons, of course).

Figure 1.3 shows what the TreeMaps execution time for simple operations is depending on. The bigger the map the longer the operations can take up to $\log(n)$ as previously mentioned. Another factor that also affects the execution time is the cost of comparisons. If the map has to traverse far down the tree a simple operation can become expensive.

1.5.3 fastutil OpenHashMap

fastutil[4] is a free java library which extends the Java Collections framework. It contains maps which have fast access and insertion operations. Its execution time depends on the same factors as the HashMap and LinkedHashMap and it is specifically encouraged on the fastutil web page to "always set explicitly the load factor, as speed is strongly dependent on the length of collision chains."[2] The map that was used in the benchmarks were the Object2ObjectOpenHashMap. I will refer to this as the OpenHashMap from throughout this paper.

The OpenHashMap uses open adressing. In open adressing no elements are stored outside of the table, so there will be no chaining like there are in the HashMap and LinkedHashMap. There are no pointers in the table but instead it computes what slots to be examined when a simple operation is used on the table. So each slot in the table is either an element or `NIL`. When collisions happen in the map the algorithm will looks for the next available slot in the hash table to store the collided key.

[3]https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html
[4]https://fastutil.di.unimi.it/

Chapter 2
Methods

This chapter is divided into three sections. First the setup of the benchmarks and the tools that were used in them. Then the benchmarks of the get operation and lastly the benchmarks of the put operations.

2.1 Setup

All the benchmarks have been run on a machine with java version JDK 11.0.12 and with an Intel eight core processor i7-10700F with 32 GB RAM.

I analyze our experimental results with linear regression, that way the trend of the data can be predicted. All the data have been analyzed and processed in Matlab to create the graphs and results which are shown in this paper.

2.1.1 Java Microbenchmark Harness

To do the benchmarks *Java Microbenchmark Harness*[1] (JMH) was used. On its github page it is explained as: "JMH is a Java harness for building, running, and analysing nano/micro/milli/-macro benchmarks written in Java and other languages targeting the JVM."[4]

All the benchmarks were run with JMH version 1.36. Each benchmark had 5 warm up iterations of one second each, five measurement iterations of 2 seconds each. They were running with one thread and five forks. The benchmark mode was set to average time.

[1]https://github.com/openjdk/jmh

String Lenght	6
Map Size	10^5
Loadfactor	0.75
Start Capacity	16

Table 2.1: Default settings for the parameters used for the maps in the benchmarks

2.1.2 Random String Array Generator

To generate collections for the benchmarks a random string array generator was built. It generates random strings of a given length based on a set seed to make sure that the sequence of strings are always the same. It only generated lower case letters from a to z. If one wished to reduce risk of collisions the generator could easily be changed to include upper case letters and/or numbers. As an example if the generator is run with the seed 100 and with string length 6 the 3 first strings will be `"stfsra" "uhqsqg" "ggmcrv"` but if the string length is changed to 4 the first 3 strings will be `"stfs" "rauh" "qsqg"`. The generator also saved a few of the random generated strings to be used later for operations like get in the benchmarks. The code was also altered during some benchmarks so that the string that got saved was never added to the array, that way it could be used in benchmarks for the put operations. I also added a function to get the number of collisions (Strings with the same hash code) in the array so that it could be measured later. The code can be read in appendix A.

The settings used as default for the different parameters can be seen in table 2.1. When one parameter is benchmarked the others will remain as these default values unless something else is mentioned.

The choice of default string length is based on the average word length in the English language. The average word length if you take into account the frequency of a word is estimated to be 4,79 letters but if you take each word into account only once the average word length is 7.60 letters[5]. Taking this into account but also the fact that collisions needed to be as few as possible in most of the benchmarks, the string length 6 was chosen.

2.2 Lookups

Here I describe how I benchmarked the get operation. Further down the Insert operations will be explained. Each following section focuses on a set of factors, for which I designed a specific benchmark.

2.2.1 Map Size and String Length

I set up a benchmark to see how map size and string length affected the execution time of the get operation. In the setup of the benchmark I created a map and filled it with n number of strings where each string was of length m. During the benchmark n was varied between $10^1, 10^2, ...10^6$, and m between 6, 10, 20, 30, 40 and 50 and the get operation was done on each variation with a previously selected string that already existed in the map. This benchmark

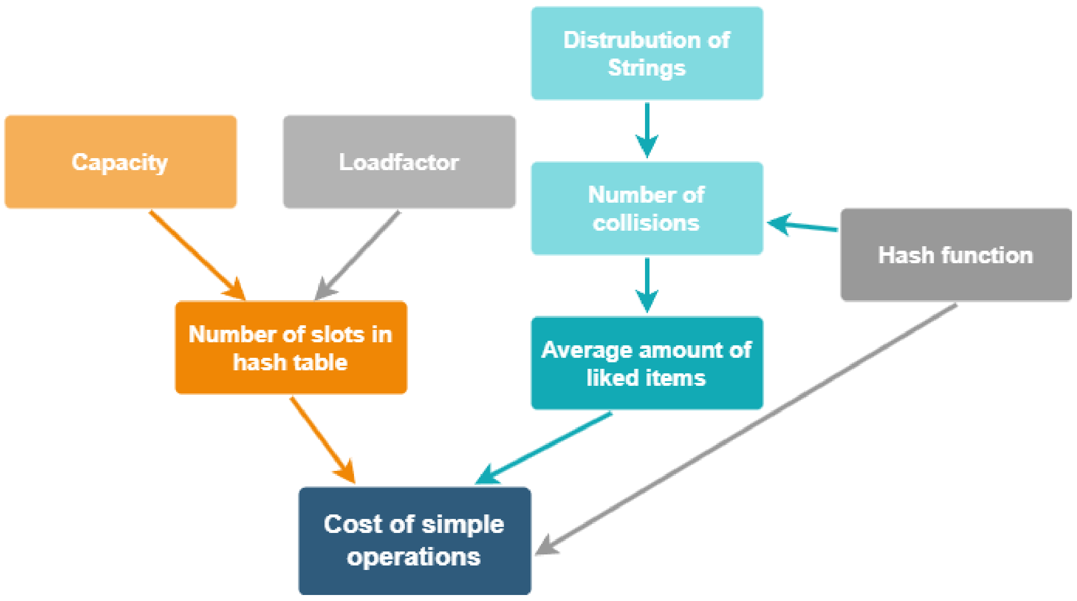

Distrubution of Strings
Capacity
Loadfactor
Number of collisions
Hash function
Number of slots in hash table
Average amount of liked items
Cost of simple operations

Figure 2.1: A Causal graph, based on figure 1.2, showing what parameters have been taking into account when benchmarking the map size and string length. The greyed out boxes are set to be their default values.

was run for the HashMap, LinkedHashMap, OpenHashMap and TreeMap. In figure 2.1 a causal graph shows what parameters that are being taken into account.

There was also an extra benchmark run for the TreeMap where the seed was changed to six different values, randomly chosen by me, together with the string length and the map size. This due to the fact that the TreeMaps operations execution time is also very dependent on where in the map the string is. If it is very far down the map it will take longer to iterate to than if it is very early in the tree.

2.2.2 Loadfactor

I set up a benchmark to see how the loadfactor in HashMap, LinkedHashMap and OpenHashMap affected the execution time of a single call to the get operation. In the setup of the benchmark the maps were filled with 10^5 elements, the string length was set to 6 and a string got picked out of the map to be used in the get operation. During the benchmark the loadfactor was varied between 0.1, 0.2,... 1. In figure 2.2 a causal graph shows what parameters that are being taken into account.

2.2.3 Collisions

To benchmark how collision affected the execution time for the get operation I created a new type of string which I called the hashString, the code for this class can be seen in Appendix A. I created a hash code function to override the java default hash code function for strings, it was set to `hashCode()%(sizeOfMap*n)` where n would be set between 0,1,... 10 and at 0 it would be set to the java default hash code for string. This function enabled me to create

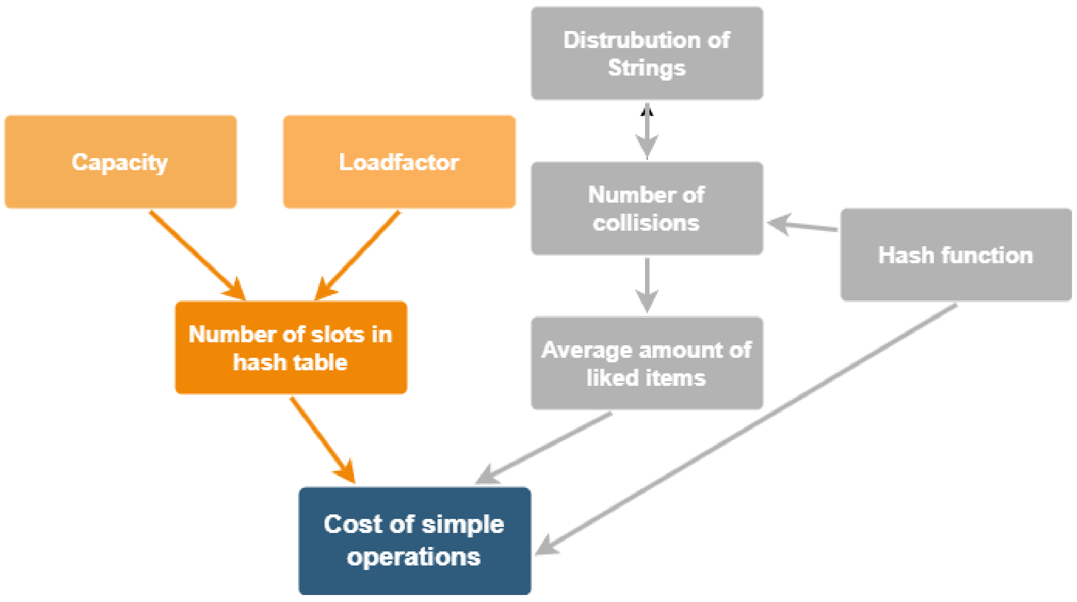

Distrubution of Strings
Capacity
Loadfactor
Number of collisions
Hash function
Number of slots in hash table
Average amount of liked items
Cost of simple operations

Figure 2.2: A Causal graph, based on figure 1.2, showing what parameters have been taking into account when benchmarking the loadfactor. The greyed out boxes are set to their default values.

collisions in the map by restricting the variation in hash codes that existed. When n was set to 2 I would observe more collisions than when n was set to 0 or to 10. Some examples of this can be seen in table 2.2. With n = 0 there was about 0.01% collision and with n = 4 there were about 50% collisions. The benchmark was then run for all the maps previously filled with 10^5 elements and string length set to 6. In figure 2.3 a representation of what parameters are changed can be seen.

Seed	n	Map Size	String Length	Collisions
100	0	100 000	6	17
100	10	100 000	6	20104
100	4	100 000	6	50296
100	0	10 000	6	0
100	10	10 000	6	1978
100	4	10 000	6	5002
253	0	100 000	6	24
253	10	100 000	6	20304
253	4	100 000	6	49815
253	0	10 000	6	0
253	10	10 000	6	1937
253	4	10 000	6	4851

Table 2.2: A table showing how many collisions happen depending on the hash function that is used. The hash function is `hashCode()%(sizeOfMap*n)` where n would be set between 0,1,... 10 and at 0 it would be set to the java default hash function for strings.

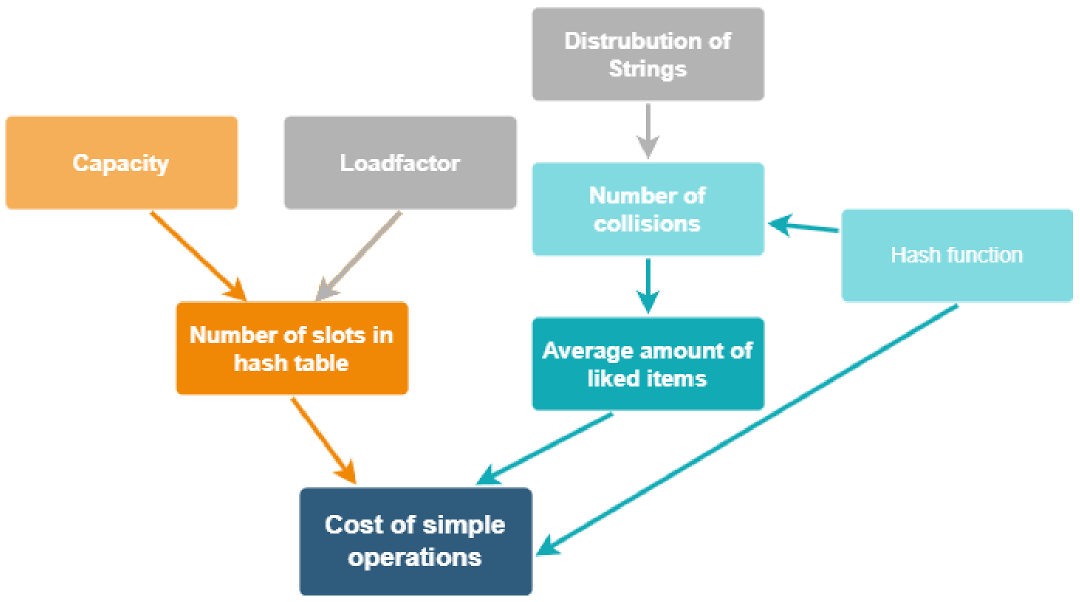

Distrubution of Strings
Capacity
Loadfactor
Number of collisions
Hash function
Number of slots in hash table
Average amount of liked items
Cost of simple operations

Figure 2.3: A Causal graph, based on figure 1.2, showing what parameters have been taken into account when benchmarking the collision rate. The greyed out boxes are set to their default values.

2.3 Inserts

Here the methods of benchmarking of the insert operations will be described. Each following section focuses on a set of factors, for which I designed a specific benchmark.

2.3.1 Map size and String Length

A benchmark was made where a string was inserted into an already filled map where the string did not already exist. The size of the maps were varied between $10^1, 10^2, ...10^6$ and the string length between 6, 10, 20, 30, 40 and 50. See figure 2.1.

There was also an extra benchmark run for only the TreeMap where the seed was changed to six arbitrarily chosen seeds, together with the string length and the map size.

2.3.2 Loadfactor

I created another benchmark where a string was inserted into a filled map. Here the loadfactor varied between 0.1, 0,2,... 1 while the map size was set to 10^5 and string length was set to 6. See figure 2.2.

2.3.3 putAll

For the putAll operation three different benchmarks were run, one to compare it to inserting all elements with a loop, one to see how the loadfactor affected its execution time and one to see how the starting capacity of the map affects the execution time, the last one will be talked about in the next section.

For the first becnhmark where I wanted to compare the putAll operation with inserting all elements with a loop I created one empty map, map1, with its default size of 16 and one map, map2, that was already filled with all the elements. I used the putAll operation on map1 and inserted all the elements of map2. For the loop version I made a foreach loop inserting each element one at a time from a previously filled array with the same elements as in map2 into map1. All these benchmarks were run with map sizes 10^5 and 10^6 and with string length 6.

For the benchmark of the loadfactor I, like above, had one map, map1, that was empty and set to its default size 16 and one map, map2, that was filled with all the elements. I then inserted all the elements from map2 into map1 with the putAll operation. During the benchmark the loadfactor was varied between 0.1, 0.2,... 1.

2.3.4 Starting Capacity

To see how the starting capacity affected the execution time when using the putAll operation a new benchmark was made. A map, map1, was created with starting capacity set to 0.1, 0.2,.. 1 and 1.25, 1.5,... 2 and 3 (10 - 100%, 125% - 200% and 300% of the size after insertion). I then had a map, map2, already filled with all the elements and used the putAll operation on map1 to insert all the elements from map2.

Chapter 3
Results

3.1 Lookups

Here the results of the get operations benchmarks will be described. Each section is a different factor that could affect the execution time.

3.1.1 Map Size and String Length

The length of the strings didn't appear to affect the execution time much at all for the HashMap and LinkedHashMap. The TreeMap and the OpenHashMap were affected more, the maps increased overall, but showed some erratic behavior. This can be seen in figure 3.1.

In the graphs in figure 3.2, that shows execution time for the get operation depending on map size, the TreeMap showed a clear increasing curve while the OpenHashMap were unstable. The HashMap and the LinkedHashMap are mostly unaffected by the map size excluding a peak at the map size of 10 000, which can be seen in figure 3.2 at 4 on the x-axis.

Both figures 3.1 and 3.2 do show that in general the TreeMap does seem to be slower overall and that the HashMap and LinkedHashMap are the fastest in general. The OpenHashMap seems to be faster than the other maps in some instances and slower than them in others.

In figures 3.3 and figure 3.4 it can be seen how dependent the TreeMap is on how far down the map a string ends up. If you have to traverse the map to the bottom it will take substantially longer than if the string is early in the tree. These figures illustrate this behavior of the TreeMap, but also that its operations still have a longer execution time.

For the HashMap, LinkedHashMap and OpenHashMap an unexpected peak of execution time happened when the map size was 10^4, this can be seen in figure 3.5. It can also be seen in this figure that the linear regression line is showing no relation to the map size.

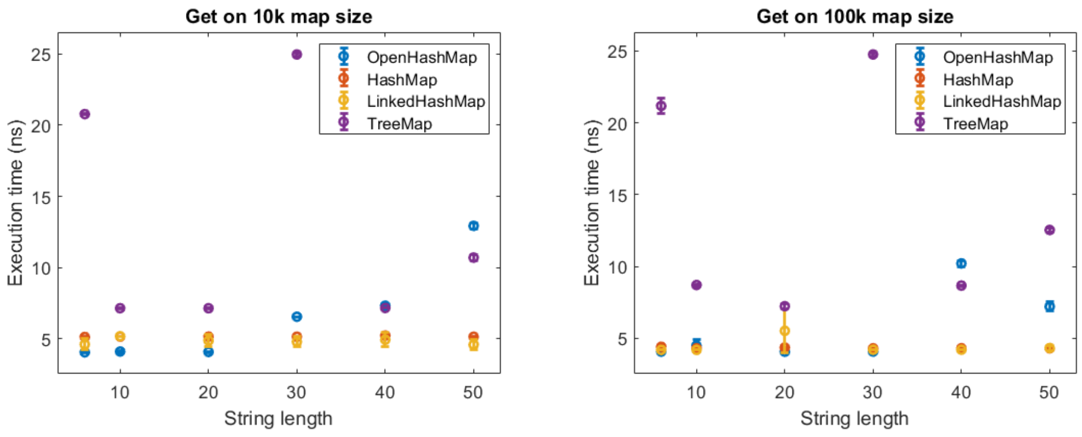


Figure 3.1: A graph showing execution time for HashMap, Linked-HashMap, OpenHashMap and TreeMap get operation, on a map that is filled with 10^5 elements and where the string that is used in the get operation already exists, depending on the size of the string. The string is varied between 6 and 10, 20,... 50.

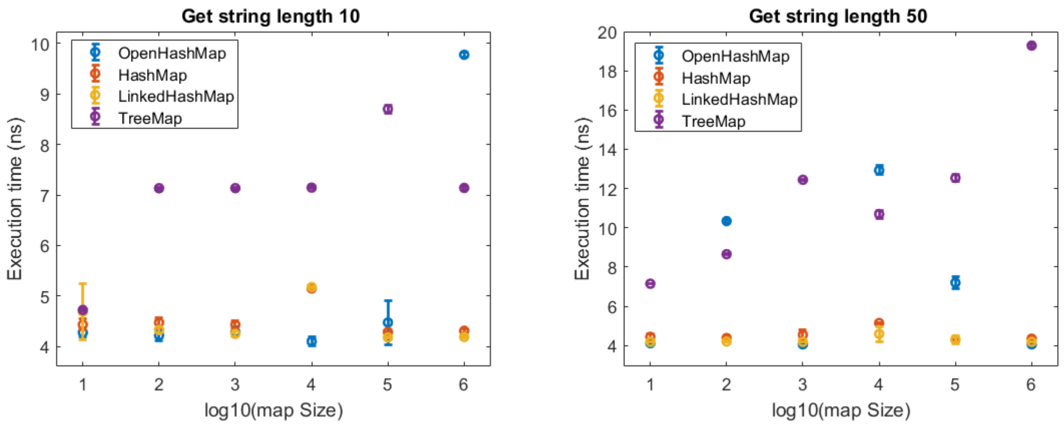


Figure 3.2: Graphs showing the execution time for the get operation, on a previously filled map where the string used in the get operation already exist, depending on the map size for HashMap, Linked-HashMap, OpenHashMap and TreeMap with the string lengths 10 and 50.

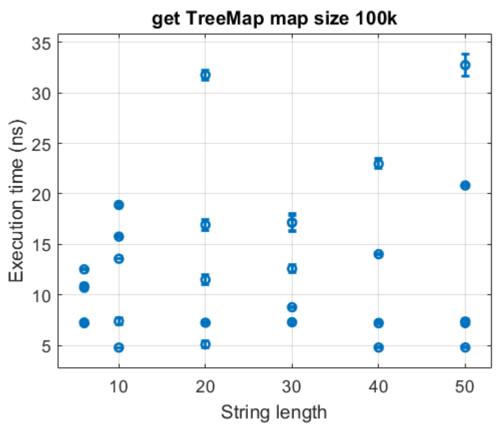


Figure 3.3: Graphs showing the execution time for the get operation, on a previously filled Treemap where the string used in the operation already exists, it was run with multiple different seeds varied over different string lengths.

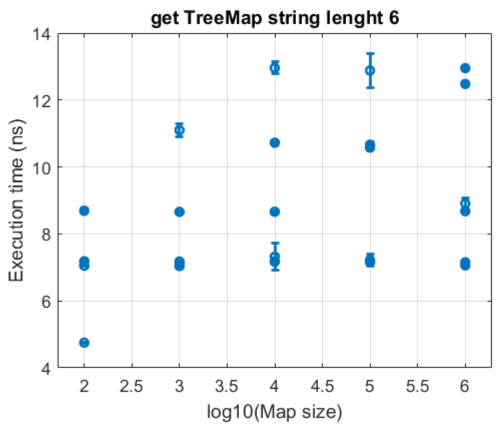


Figure 3.4: Graphs showing the execution time for the get operation, on a previously filled Treemap where the string used in the operation already exists, it was run with multiple different seeds varied over different map sizes.

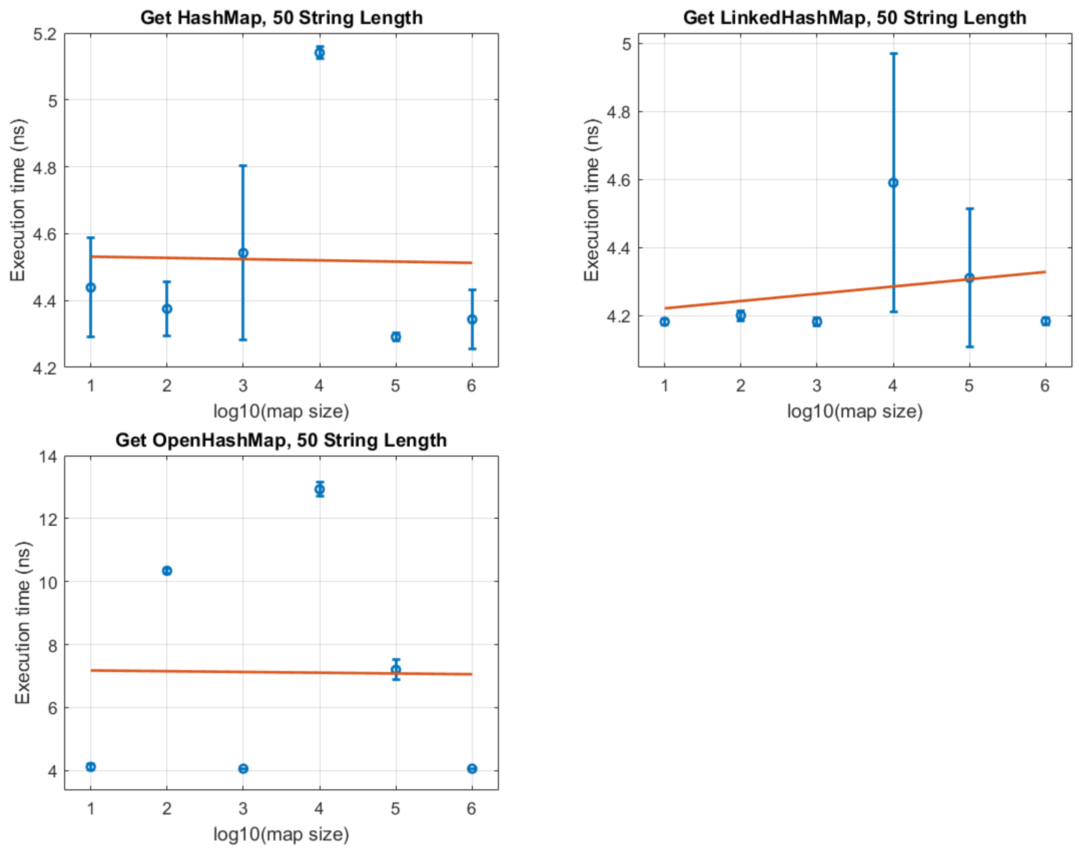


Figure 3.5: Graphs showing the execution time for the get operation, on a previously filled map where the string used in the operation already exists, depending on the map size for HashMap, LinkedHashMap and OpenHashMap. At the map size of 10 000 (at 4 on the x-axis) a spike in execution time can be seen on all the graphs for all the maps. In all the graphs the red horizontal line is a linear regression analysis of the data.

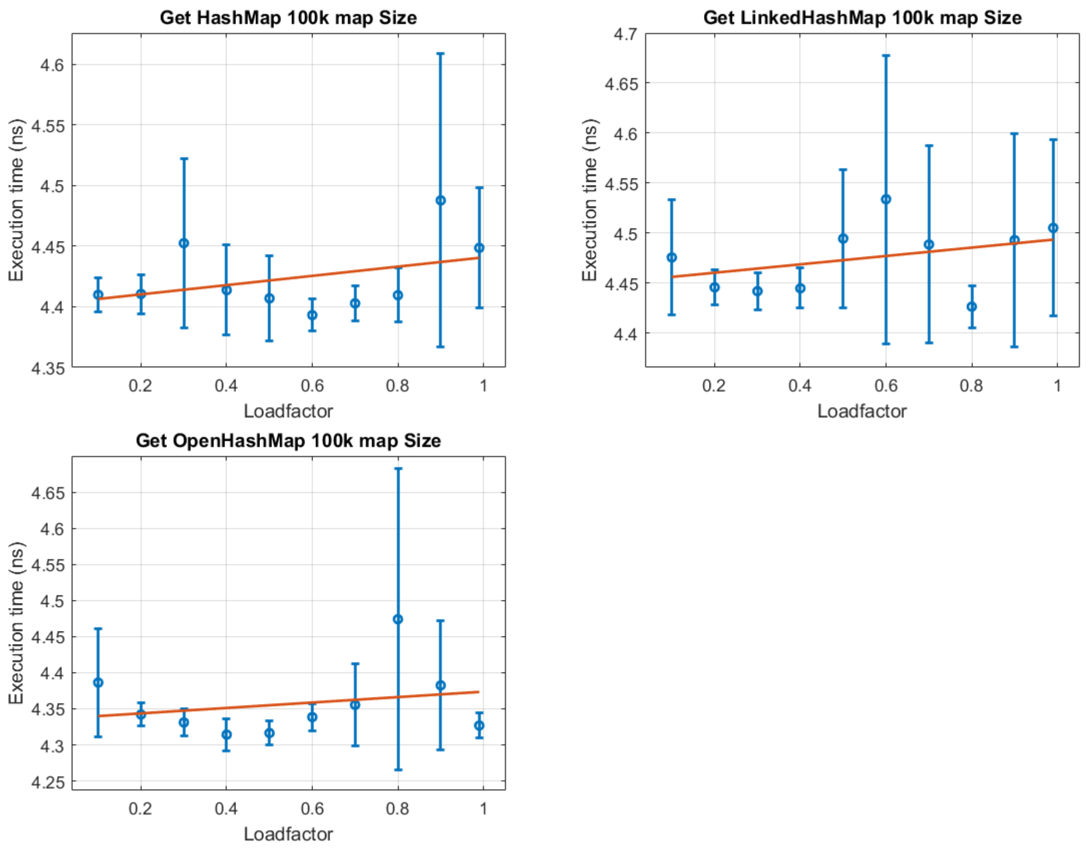


Figure 3.6: A graph showing execution time for the get operation, on a map previously filled with 10^5 elements where the string used already exists, for HashMap, LinkedHashMap and OpenHashMap depending on the loadfactor. In all the graphs the red horizontal line is a linear regression analysis of the data. A slight upward trend can be seen for all three maps.

3.1.2 Loadfactor

Figure 3.6 shows that the loadfactor does increase the execution time for the lookup operation. A bigger loadfactor leads to a longer execution time. The OpenHashMap seems to be slightly less affected than the HashMap and LinkedHashMap.

3.1.3 Collisions

Figure 3.7 shows that the execution time is heavily dependent on the amount of collisions occurring. So having a hash function that to the greatest extent avoids collision will decrease the execution time. The default hash code is used and compared to a hash code that instead is set to `hashCode()%(sizeOfMap*n)`. A smaller n will lead to more collisions except when n = 0, then the default java hash function is used.

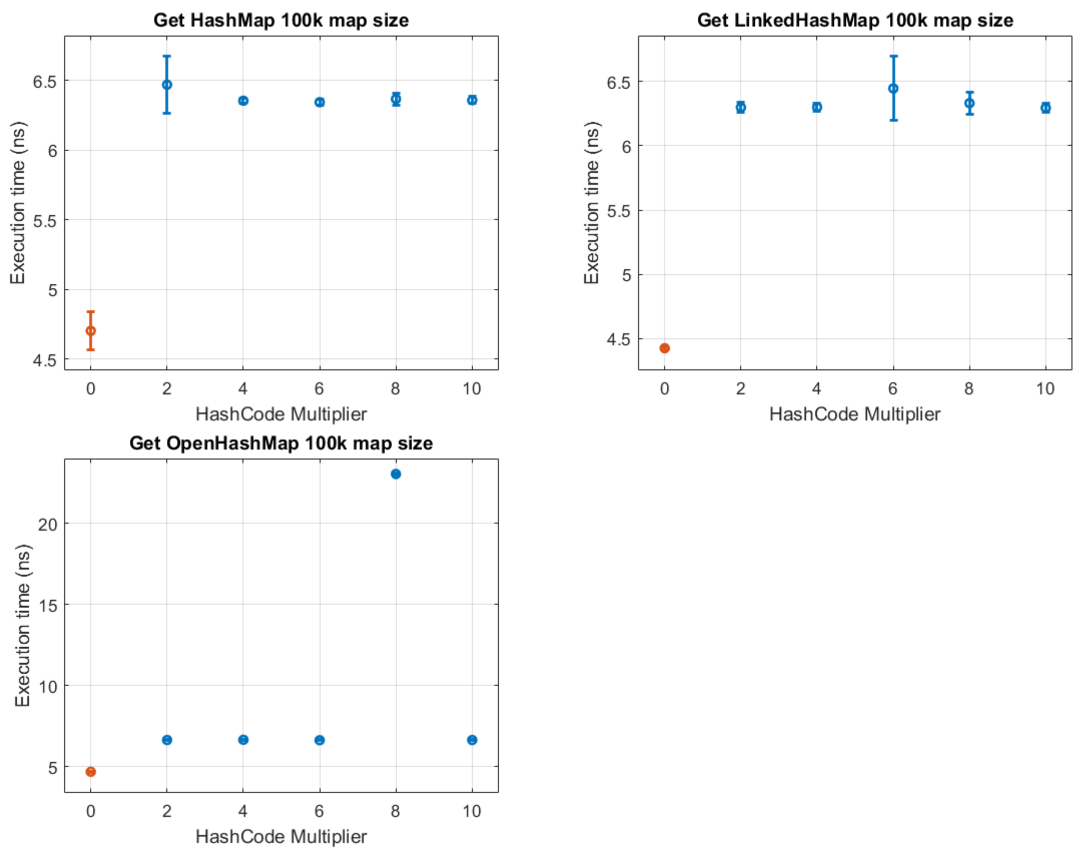
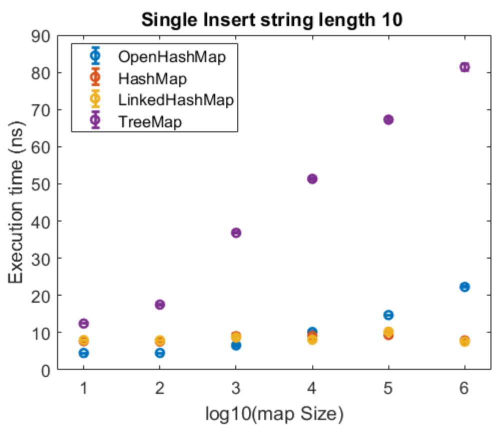


Figure 3.7: A graph showing execution time for HashMap, LinkedHashMap and OpenHashMap get operation depending on the hash code that is being applied on the string. The red point is the default hash function, that java uses for string, while the blue points are the hash function that is `hashCode()%(sizeOfMap*n)`, where n is the x-value you see on the graph.



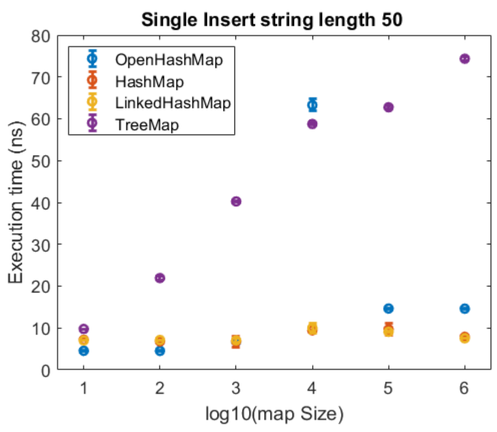


Figure 3.8: Graphs showing the execution time for the put operation on previously filled HashMap, LinkedHashMap, OpenHashMap and TreeMap depending on their map sizes. The graph show the logarithmic values for the map sizes. The original sizes were $10^1, 10^2, \ldots 10^6$.

3.2 Insert

Here the results of the put operations benchmarks will be described. Each section is a different factor that could affect the execution time.

3.2.1 Map Size and String Length

In figure 3.8 it can be seen that the execution time of the HashMap and the LinkedHashMap are mostly unaffected by the size of the map. The execution time of the TreeMap is as expected heavily affected and an almost straight line corresponding to $\log(n)$, with n being the map size, can be seen. The OpenHashMap has quite unstable results but still performs better than the TreeMap overall. And it seems to perform better than all the other maps on smaller map sizes ($<10^4$) but perform worse than the HashMap and the LinkedHashMap on bigger map sizes. . When looking at how the string length affected the execution time for the maps in figure 3.10 it again seems like the HashMap and the LinkedHashMap are generally unaffected by it while it seems to show a growing trend for the TreeMap and the OpenHashMap.

Figure 3.9 and figure 3.11 shows how dependent the TreeMap is on how far down the tree a string ends up. If it has to traverse the map to the bottom it will take substantially longer than if the string is early in the tree. These figures give a scope of how dependent the TreeMap is of that but also that its operations still perform worse than the other maps.

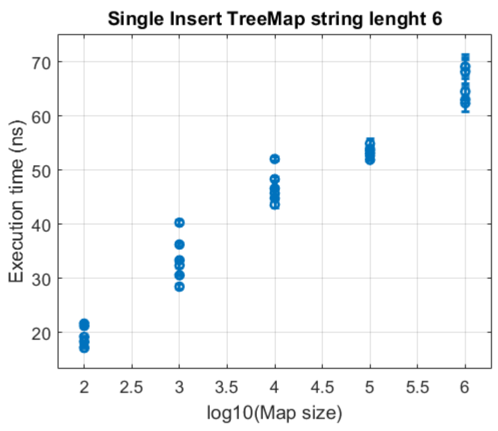


Figure 3.9: Graphs showing the execution time for the put operation, on a previously filled Treemap where the string used in the operation already exists, it was run with multiple different seeds varied over different map sizes.

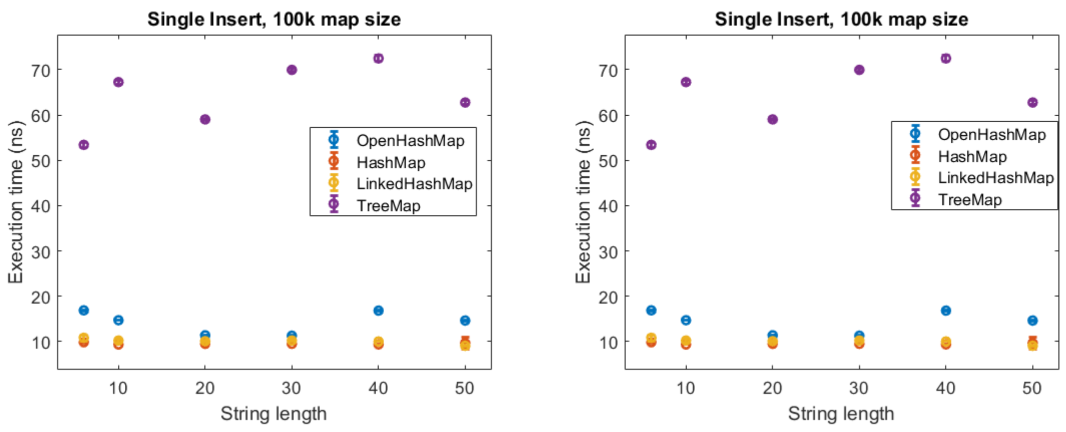


Figure 3.10: Graphs showing the execution time for the put operation on previously filled HashMap, LinkedHashMap, OpenHashMap and TreeMap depending on their string length.

3.2.2 Loadfactor

In figure 3.12 it can be seen how the loadfactor affects the execution time for the put operation for both the HashMap, LinkedHashMap and the OpenHashMap. For the put operation for the HashMap and the LinkedHashMap a higher loadfactor resulted in a higher execution time while for the OpenHashMap a higher loadfactor did the opposite and reduced the execution time.

3.2.3 putAll

The putAll operations benchmarks show that there is a difference between using the putAll operations and making a loop to insert all elements. This can be seen in fig 3.13 where it can be seen that for the LinkedHashMap it is less costly to use the putAll operation while for that HashMap and the OpenHashMap it varies a bit which is better.

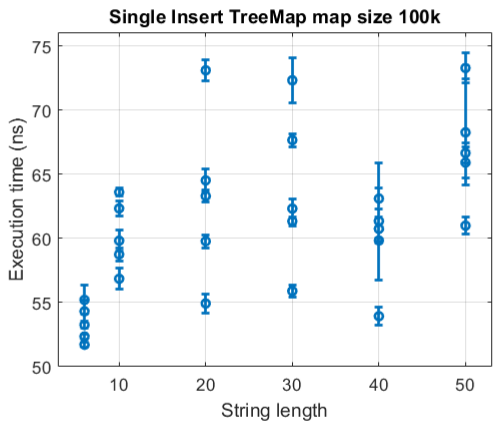


Figure 3.11: Graphs showing the execution time for the put operation, on a previously filled Treemap where the string used in the operation already exists, it was run with multiple different seeds varied over different string lengths.

Inserting many strings at once made the HashMap behave unexpectedly compared to what have been previously seen when the loadfactor is changed. As the documentation of the HashMap describes and as can be seen in previous benchmarks an increased loadfactor should lead to an increased execution time.However with the putAll operation on the HashMap it is the opposite, a higher loadfactor reduces the execution time. This can be seen in fig 3.14.

3.2.4 Start Capacity

As can be seen in fig 3.15 the execution time increases when the starting capacity increases for the putAll operation on the HashMap and LinkedHashMap. For the OpenHashMap it does more seem to have a downwards trend with the execution time reducing when the starting capacity increases.

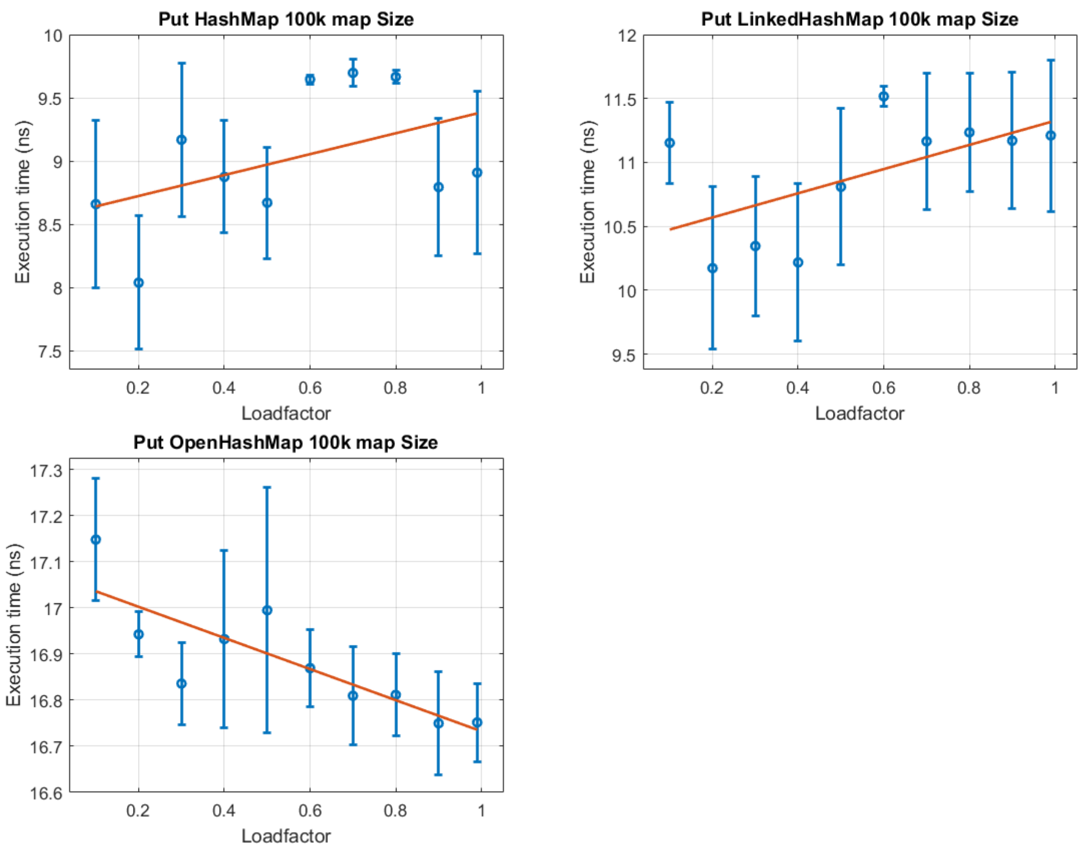


Figure 3.12: Graphs showing the execution time for inserts into previously filled HashMap, LinkedHashMap and OpenHashMap in relation to their loadfactor. The red line is a linear regression analysis of the data.

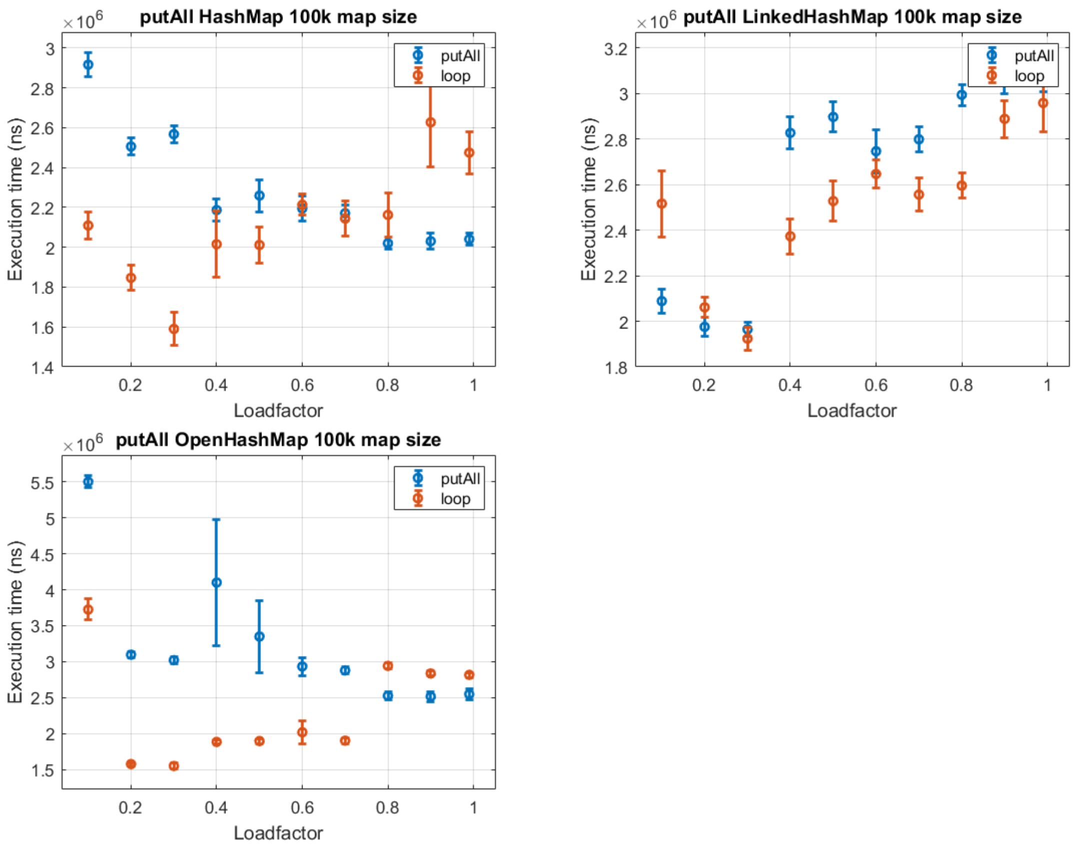


Figure 3.13: Graphs showing the difference, between the putAll operation and using a for loop, in execution time when adding 10^5 strings at once into the HashMap, LinkedHashMap and OpenHashMap.

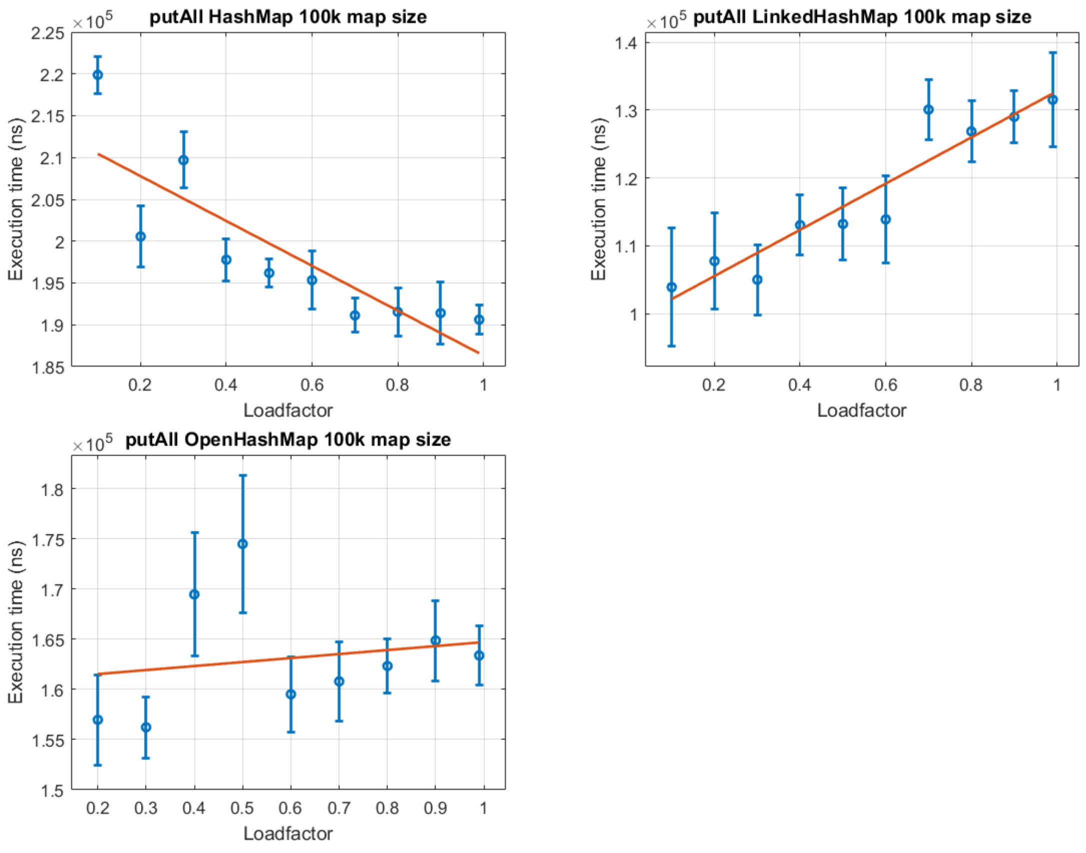


Figure 3.14: Graphs showing how the execution time is dependent on the loadfactor when using the putAll operation. The lines are linear regression analysis of that the trend is in the data.

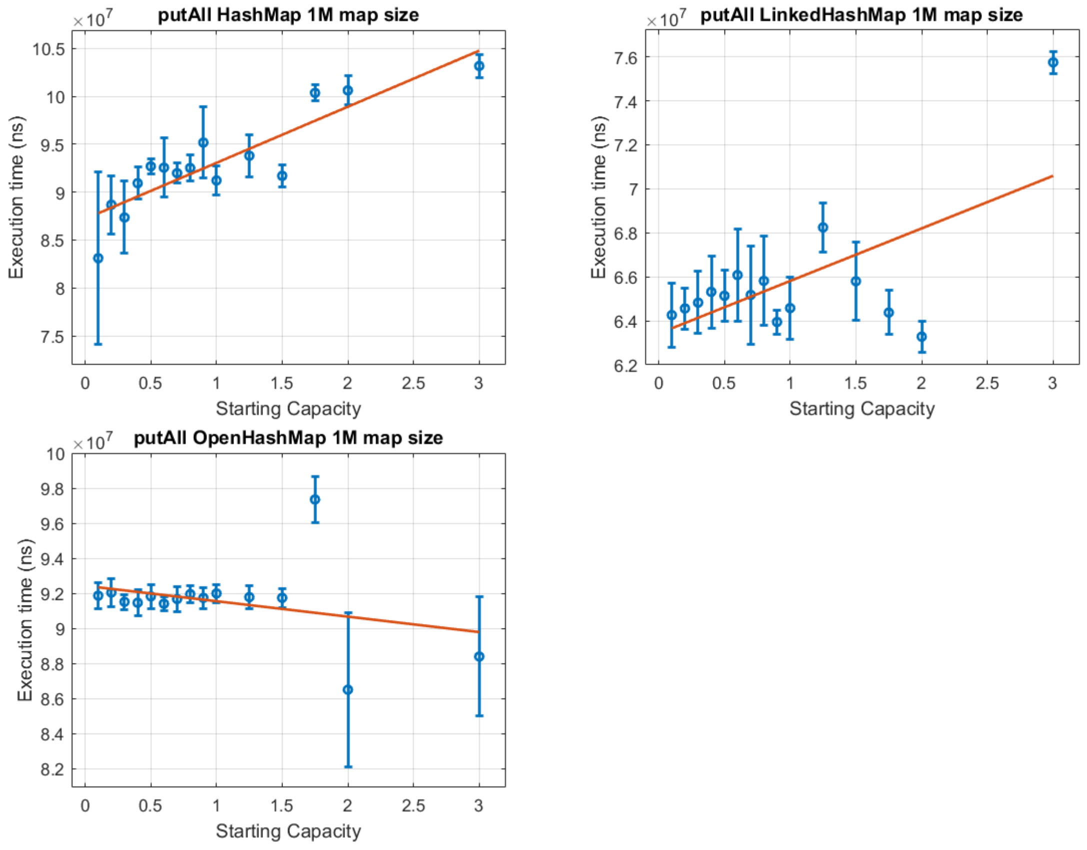


Figure 3.15: Graphs showing the execution time for the putAll operation when the starting capacity is altered on the HashMap, LinkedHashMap and OpenHashMap. The starting capacity is a factor of the end size, 0.1 is 10% of full capacity and 2 is 200% of full capacity.

Chapter 4
Discussion

4.1 Overall Performance

My benchmarks show that the TreeMap behaves as both the books *Introduction to Algorithms* and *The Algorithm Design Manual* and the Java documentation. They describe the TreeMap with a $\log(n)$ time cost for the simple operations and the cost not being affected by any other parameter. The TreeMap falls behind the others when it comes to execution time for its simple operations. The only time the TreeMap has been able to compete is when the map size was <100. It is unaffected by collisions since it uses comparisons when traversing down the map and could thus potentially perform better when there are many in the map.

The OpenHashMap have been a quite unstable with its execution time for simple operations as can be seen in the figures 3.1, 3.2 and 3.10. It generally preforms the same as the HashMap and the LinkedHashMap. *The Algorithm Design Manual* states that for the OpenHashMap performance gets worse with a higher loadfactor. From what I have observed this is true when doing lookups but not when doing insertions. Instead it showed that the bigger the loadfactor was the faster execution time, see figure 3.12.

When using the putAll operations the execution time for the OpenHashMap was much less affected by both the loadfactor and starting capacity than the HashMap and LinkedHashMap, see figure 3.15 and 3.14. Over all the OpenHashMap performed slightly worse than the HashMap and the LinkedHashMap for simple operations but when using the putAll operation it performed slightly better.

The LinkedHashMap and the HashMap did the best in general out of all the maps. They were unaffected by map size and string length which is consistent with what *Introduction to Algorithms* states of having a complexity of $O(1)$. The loadfactor gave both the maps the same results for simple operations, bigger loadfactor lead to longer execution time. Collisions heavily affects execution time and should to the greatest extent be avoided. Both the result for the loadfactor and the result for the collisions were consistent with the Java documentation.

4.2 The putAll Operation

When inserting many elements at once one would expect the putAll operations and a for loop inserting the elements one at a time, would take approximately the same amount of time, but that is not what happens. As can be seen in figure 3.13 for the HashMap and the OpenHashMap the putAll operations performs better with a smaller loadfactor but the loop does better with a bigger loadfactor. For the LinkedHashMap the putAll operation performs better overall.

The documentation of the HashMap states "*An instance of HashMap has two parameters that affect its performance: initial capacity and load factor... As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put).*" while the documentation for the LinkedHashMap states "*A linked hash map has two parameters that affect its performance: initial capacity and load factor. They are defined precisely as for HashMap. Note, however, that the penalty for choosing an excessively high value for initial capacity is less severe for this class than for HashMap, as iteration times for this class are unaffected by capacity.*". So we would expect a smaller loadfactor to give a shorter execution time and that the HashMap and LinkedHashMap behaved the same. But when the putAll operation was benchmarked the results were contradicting to what the documentation is stating. The HashMap and the LinkedHashMap behaved opposite to each other, a bigger loadfactor for the HashMap gave the putAll operation a faster execution time, while for the putAll operation on the LinkedHashMap got a longer execution time when the loadfactor was increased. This can be seen in figure 3.14.

The starting capacity then affected the execution time in the same way for both the HashMap and the LinkedHashMap when using the putAll operation, a bigger starting capacity lead to a longer execution time. This is again contradicting what the documentation states. Based on the documentation, I would expect a bigger starting capacity to lead fewer rehashes and therefore reducing the execution time. The documentation encourages the user to minimize the number of rehashes as it states: "*The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations.*". So what would be expected is that the execution time would be at its best when the starting capacity is bigger than the expected number of entries. However as can be seen in figure 3.15 this is not the case. It instead shows that the starting capacity brings the shortest execution time when the starting capacity is smaller than the expected number of entries.

The authors of *The Algorithm Design Manual* write: "*How big should the table be?: With bucketing, m should about the same as the maximum number of items you expect to put in the table. With open addressing, make it (say) 30% larger or more.*". But this is not what is giving the best time performance, as can be seen in figure 3.15 not one of the 3 maps follow the expected behaviour.

It is possible that this behaviour for the putAll operations is because the map computes the number of entries that is going to be inserted and therefore resizes directly so that all the entries can fit, instead of resizing multiple times as it otherwise would if the number of entries are bigger than the starting capacity.

4.3 Lookup at 10 000

Based on the complexity if lookups for the HashMap, LinkedHashMap and OpenHashMap we would expect a constant cost unrelated to the size of the map. However there was an outlier that resulted in a peak at the map size 10 000 when using the lookup operation for these maps, this can be seen in fig 3.5. This outlier have been persistent through out all the benchmarks I have done. I have tried using a different seed for the string generator to see if that has something to do with it, changing the string length and changing the loadfactor. But in all cases the execution time is noticeably higher at the map size of 10 000. I have been unable to find a reason for this peak.

Chapter 5
Threats to Validity

My PC could have been interfering during the benchmarks, there is no guarantee it didn't have anything else running in the background. If a background task starts in the middle of a benchmark running it is more likely to affect only a few samples, giving those samples results that make them look a lot worse than they might be.

I have been restricted in the parameters I used. I have mostly been using string length of 6 or 50, map size of 10 000, 100 000 and 1 000 000, default loadfactor of 0.75 and only 4 the different maps HashMap, LinkedHashMap, TreeMap and OpenHashMap.

I have not varied the seed in the random array generator. Varying the seed makes sure the data set does not affect the benchmark results in the same way every time. I have when rerunning some benchmarks changed the seed that was used. In these benchmarks I got the same results as with previous seed, meaning that changing the seed for these benchmarks had no effect. The benchmarks that was rerun with a new seed was: Map size and string length for both lookups and insertions, loadfactor for both lookups and insertions. However it is not done in a consistent way to exclude that the data set created certain patterns in the results.

I have not taken into account the cost of the hash function when getting the hash code for a string. This is due to the fact that java strings cache the result of the hash function and therefore only does it once per simple operation.[1] The time for executing the hash function once is very small in comparison to the execution time as a whole for one simple operation.

[1]https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/lang/String.java

Chapter 6
Future Work

I found a surprising result showing an outlier, with a execution time increase at the map size 10 000 when using the lookup operation for the HashMap and LinkedHashMap. Exploring this more and finding out why this is happening could be interesting.

I've been restricted in how big maps and how long strings I can store. At a map size of 5 000 000 and string length of 50 my system started to slow down and there was a obvious strain on my pc. So having a deep dive on very big collections could be an interesting given that you have a system that can handle it. It would also suggest to run benchmarks with more data points, for example I ran the loadfactor for 0.1, 0.2,... 1. But running it for 0.01,0.02,... 1 would give more data.

The remove operation could also have been a potential addition to this paper, due to the fact that it is often used.

One thing about JMH when benchmarking with it is that it constantly runs the tests in the same order. So the same thing will be at the start, end and middle of the run every time. Introducing random ordering of the benchmarks could increase the validity of the results.

While benchmarking the loadfactor for the OpenHashMaps put operaton I was expecting it to show and increased execution time with an increased loadfactor. This was not the case and it instead showed the opposite with a reduced execution time with a bigger loadfactor. It could be that the benchmarking is broken and it could fix it by rerunning this benchmarks with randomizing the order of the runs. Currently it runs loadfactor 0.1 first then 0.2 up to 1, I suggest trying to run them in a random order.

There is a chance when benchmarking the put operation on the HashMap, LinkedHashMap and OpenHashMap that a single insertion can trigger a rehash of the map. I would suggest benchmarking the the rehash method to see how big of an affect it does have on the execution time.

I have made one benchmark of the collision in this paper. However running some more benchmarks to try to determine if the cost of collisions is different for different maps.

Chapter 7

Conclusion

What does affect the execution time for simple operation on maps? What have been concluded in this paper is that it 1, depends on the map and 2, depends on what set of data you have. The HashMap, LinkedHashMap and the Open HashMap all have a lot of parameters that can affect the execution time for their simple operations. As the documentation stated I have found that the loadfactor does in general increase the execution time the bigger it is for simple operations (not true in all cases, this can be seen in figure 3.12 where the opposite is true for the OpenHashMap). Contradicting to the documentation I have found that increasing the starting capacity increases the execution time for the HashMap and LinkedHashMap when using the putAll operation. But for the OpenHashMap when using the putAll operation it behaves as we would expect from the documentation with having a shorter execution time when the starting capacity increases. What the data set looks like can also affect the execution time for simple operations notably. As the documentation stated collisions have a detrimental effect as can be seen in figure 3.7. This can be controlled somewhat with the hash function (but likely you will only use the default one Java uses) or by changing the data set. Other than that, the string length or the map size does not affect performance as is expected from the documentation. The TreeMap behaves as the documentation describes and can only be manipulated by changing the data set. It's not affected by collisions since it always does comparisons while traversing the map and the string length have a small impact on the execution time due to the cost of comparisons.

Is the current documentation of the maps correct? For simple operations on the HashMap, LinkedHashMap and TreeMap the answer is yes. They behave as the documentation have described it in regards to all the benchmarks I have run. The map size only affects the TreeMap. A increased loadfactor leads to and increased execution time. Collisions increase the execution time for the HashMap and the LinkedHashMap. For the OpenHashMap when using simple operations it behaves as we expect from the documentation, same as the HashMap and LinkedHashMap. However there was an exception when using the put operation while benchmarking the loadfactor. There we expect an increased execution time with an increased loadfactor, but what the benchmark showed was that it got a reduced execution time with an increased loadfactor.

For the putAll operation the documentation tells us that a starting capacity bigger than the collection size should give a better execution time. It also tells us that a smaller loadfactor should give us a shorter execution time and that the HashMap and LinkedHashMap should behave the same in regards to the loadfactor. But for the HashMap and the LinkedHashMap this is not what we see. The starting capacity gave a shorter execution time when it was smaller than the collection size. The HashMap and LinkedHashMap did not behave the same regarding the loadfactor and the HashMap hand a shorter execution time when the loadfactor got bigger. When using the putAll operation on the OpenHashMap it did behave as the documentation describes it.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. Massachusetts Institute of Technology, 2009.

[2] https://fastutil.di.unimi.it/. fastutil: Fast & compact type collections for java. `https://fastutil.di.unimi.it/` Accessed: 2023/03/20.

[3] https://github.com/openjdk. Openjdk string.java. `https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/lang/String.java` Accessed: 2023/03/20.

[4] https://github.com/openjdk/jmh. Java microbenchmark harness (jmh). `https://github.com/openjdk/jmh` Accessed: 2023/01/20.

[5] Peter Norvig. English letter frequency counts: Mayzner revisited or etaoin srhldcu. `http://norvig.com/mayzner.html` Accessed: 2023/03/17.

[6] Steven S. Skiena. *The Algorithm Design Manual, Second Edition*. Springer-Verlag London Limited, 2008.

Appendices

Appendix A
Random Array Generator & HashString Code

```
public class RandomArrayGenerator {

    private ArrayList<String> list;
    private int size;
    private RandomStringGenerator gen;
    public ArrayList<String> savedWords;

    public RandomArrayGenerator(int wordlength,
                                int size){
        this.size = size;
        list = new ArrayList<String>();
        gen = new RandomStringGenerator(wordlength);
        savedWords = new ArrayList<String>();
    }

    public ArrayList<String> generateArray(){
        String word;
        int mod = (int) Math.ceil(size/3);
        if(size < 1){savedWords.add(gen.generateWord());}
        for(int i = 0; i < size; i++){
            word = gen.generateWord();
            if(i%mod == 0){
                savedWords.add(word);

            }
                list.add(word);
        }
        return list;
    }

    public int getNumberOfCollisions(){
        int col = 0;
        for(int i = 0; i< list.size(); i++){
            for( int j = i+1; j<list.size(); j++){

                if(list.get(i).hashCode() ==
                 list.get(j).hashCode()){
                    col++;
                }
            }
        }
        return col;
    }

    public ArrayList<String> getSavedWords(){
        return savedWords;
    }
}

public class RandomStringGenerator {
    private int wordLength;
    private int leftLimit = 97; // letter 'a'
    private int rightLimit = 122; // letter 'z'
    private long seed = 253;
    private Random random;


    public RandomStringGenerator(int length){
        wordLength = length;
        random = new Random(seed);
    }

    public String generateWord(){

        StringBuilder buffer = new StringBuilder(wordLength);
        for (int i = 0; i < wordLength; i++) {
            int randomLimitedInt = leftLimit + (int)
              (random.nextFloat() * (rightLimit - leftLimit + 1));
            buffer.append((char) randomLimitedInt);
        }
        String word = buffer.toString();
        return word;
    }
}
```

```
public class HashString {
    public String word;
    public double loadf;
    public int size;

    public HashString(String s,int size, double load){
        word = s;
        loadf = load;
        this.size = size;
    }

    @Override
    public boolean equals(Object o){
        return o.toString().equals(word);
    }

    @Override
    public int hashCode(){
        return word.hashCode()%(int)((size)*4);
    }

    @Override
    public String toString(){
        return word;
    }

}
```