Performance evaluation of Map implementations in Java, Python and C#

KRISTIAN NEDELCHEV, University of Twente, The Netherlands

Time and memory efficiency in data structures is an important topic in computer science as it can have a great impact on the size of the problems that could be solved as well as on the user experience. This research aims to show if there is difference in performance of the Map data structure implementations in three of the most popular high-level languages according TIOBE index[1], namely Python, Java and C#. The research examines the memory usage and the execution time, for solving the membership problem of different sizes, of Python's *Dictionary*, Java's *HashMap* and C#'s *Dictionary*. However, this research is limited to 32-bit integers for both *keys* and *values* in the respective data structures. The research concluded that Java's *HashMap* and C#'s *Dictionary* have similar memory usage and execution times, with Java having a slight edge in terms of execution time and C#'s implementation being slightly more memory efficient. At the end, known implementation differences among the data structures are discussed in the light of the results of this research, and conclusions are drawn.

Additional Key Words and Phrases: Python, Java, C#, Performance Evaluation, Associative Array, Map, Dictionary

1 INTRODUCTION

It is very important for computer software that must be scalable to be built with efficiency in mind. Even small improvements in the performance of a single data structure operation can add up to a big difference in performance when it comes to problems of a large size. Furthermore, in the era of mobile devices, such as smartwatches and smartphones, if the software that runs on them does not utilize their resources in an efficient way, that can have a negative impact on the user experience and thus, detrimental for the success of the application[2]. For example, a mobile application that does not utilize CPU and memory resources efficiently, leads to higher battery consumption that is experienced as reduced battery life by the user and potentially a device that is running hot[5].

In computer science, there is the concept of an abstract data type called associative array[2], that is also known as map or dictionary, that represents a collection of *key-value* pairs. While there are simple implementations of the associative array with time complexity for lookup and deletion operations of $O(n)$ that is suitable for small number of mappings, such as the association list (also known as alist), the two most common implementations are hash map (also known as hash table) and search tree, with hashmaps having better time complexity in the average-case defined by $O(1)$[1, 6, 8]. As such an important data structure in computer programming, in this research we are going to compare the time and memory efficiency of its implementations in three programming languages. According to the TIOBE index, among the top five most popular languages, as of November 2022, are Python, Java and C#. All these languages are so-called high-level languages that have a lot of similarities among them, such as control flow statements, loops, data structures, and programming paradigm and can be used for the development of the same type of applications. For instance, all three could be used in the development of web applications, mobile applications, video games and so on. By comparing the performance of the available implementations of the map data structure in each of them, this research will give insight into whether one of these languages is more suitable, in terms of performance, for writing software that contains map data structures of large size.

In Section 2, we will look at the benchmarks that are in the heart of our experiment closely and discuss the properties that they are trying to measure. In Section 3, we will discuss the Map implementations that were selected from each language, the design of the experiments and finally the results of the experiments. At the end, in Section 4, a summary of the findings in this paper and its general ideas can be found.

2 BENCHMARK METHODOLOGY

Definition 2.1. (Membership Problem). A universe $U = \{0, 1, 2, .., u-1\} \subset N$ and a set $S \subseteq U$ where $\|S\| = n$ are given.

To determine which language has the best performing map data structure, we have designed and implemented three identical benchmarks[3] in each of the languages that evaluate different aspects of their performance under reproducible circumstances. Each benchmark is executed on data structures of size $\boldsymbol{n}$, for 10 different $\boldsymbol{n}$ values. The results of each language in each benchmark are compared against those of the other two by means of graphs to determine if there is difference in their performance.

2.1 Layout of the benchmarks

The first benchmark evaluates the memory usage of the data structure by filling it up with a pre-generated set of $\boldsymbol{n}$ integer *key-value* pairs, called the setup set. We generated 10 unique setup sets with different size $\boldsymbol{n}$ that is increasing with each consequent benchmark run and reaches peak value of $[2^{20}]$ in the 10th and final run. For the rest of the paper, we will refer to this benchmark as a memory usage benchmark. We did not measure the time that the data structures needed to store the setup sets because we used different libraries in each of the languages to load the data from a JSON file into the data structures. If we measured the setup time the results wouldn't only expose performance differences among the languages and their map implementations but also in the performance of the JSON parsing

[1]https://www.tiobe.com/tiobe-index/

[2]https://en.wikipedia.org/wiki/Associative_array#Self-balancing_binary_search_-trees

[3]*The source code of the data set generating script, the source code of the benchmarks and the script used to create the graphs, as well as the data sets and the results can be found on https://github.com/itsall0sN1s/map-performance-eval

TScIT 38, February 3, 2023, Enschede, The Netherlands

libraries that we couldn't possibly account for.

The other two benchmarks test the performance of the data structures on the membership problem as defined in Definition 2.1. The first of the two, evaluates the data structure performance on the static membership problem by performing $[2^{20}]$ *isMember* queries and will be referred to as the *static problem benchmark*. This benchmark is based on a pre-generated list of integers, called *static problem set*, that are not guaranteed to be unique and that may or may not be existing *keys* in the setup set. There are in total 10 unique pre-generated *static problem sets* for each run of the benchmark. The second one, evaluates the performance on the dynamic membership problem instead, by performing $[n/3]$ *isMember*, *insert* and *delete* queries that are randomly interleaved, and will be referred to as *dynamic problem benchmark*. For the implementation of this benchmark, a list of tuples for the *isMember* and *delete* operations and triples for the *insert* operation is used. This list will be referred to as *dynamic problem set* for the rest of the paper, of which 10 such sets were generated for each run of the benchmark that are unique. The first element in each tuple defines the operation to be performed on the data structure, the second is the *key* on which the operation will be performed. The first two elements in the triples serve identical purposes as those in the tuples and the third extra element is the *value* which will be inserted in the dictionary with the corresponding *key*.

2.2 Generation of Setup, Static and Dynamic problem sets

The *setup*, *static problem* and *dynamic problem sets* were generated independently in Python with the SciPy[4] and NumPy[5] packages. To ensure that these sets were truly random to each other, we generated a unique seed for each set with NumPy for the random number generator of SciPy. The generated *keys* for the setup set range from 1 to $\boldsymbol{n}$ and are unique by definition. They were scrambled so that the *keys* in the resulting data structure do not follow numerical order, as it most likely be the case in a real-world application. The generated values for those *keys* are non-unique random numbers that are between $[10^5]$ and $[3x10^5]$. Both, *keys* and values, follow the uniform distribution as, again, we expect that to be the case with most real-world data sets. The number of *delete*, *insert* and *isMember* operations in the *dynamic problem set* is equal, and their order is completely randomized.

2.3 Benchmark design choices

Below, we will go through the specific design choices that we made for the generated data sets that are the heart of our benchmarks.

2.3.1 Dynamic problem benchmark. In the *dynamic problem set*, the *keys* found in the tuples for the *delete* operation are guaranteed to be present while those in the triples for the *insert* operation are guaranteed to not be in the initialized data structure that is being tested. This property of the *dynamic problem set* is very important because if the entry for a *key* not present in the data structure is requested to be deleted or if an entry for a *key* already exist, in the case of the *insert* operation, the benchmark will effectively measure not how long it took it to actually remove it or add a new entry but the time that it took the data structure to search for such entry, which is essentially what *isMember* does. In contrast, only $[2/3]$ of the *keys* for the *isMember* queries are guaranteed to be contained in the data structure. The reason behind this design choice is that a *key* not contained in the data structure represents the worst-case scenario for the *isMember* operation, regardless of its implementation, and knowing how each Map implementation performs relative to the other two could be of a big importance in real-world applications. To guarantee the properties of the *dynamic problem set* discussed above and for the sake of simplicity in generating those sets, the size of each such set is exactly equal to the size of the corresponding setup set of that benchmark run.

2.3.2 Static problem benchmark. In contrast to the *dynamic problem sets*, the *static problem sets* have the same size of $[2^{20}]$ for each benchmark run. The choice of $[2^{20}]$ as the size of the *static problem benchmark* sets was previously concluded to be big enough to effectively evaluate performance of data structures and expose differences in such, but also not too big to hinder testing[4, 7].

2.4 Performance measurements

Below we will go through the performance measurements that we are going to evaluate in this research, namely the memory usage and execution time, and the technical specifics of how they were measured in each language.

2.4.1 Memory usage. Memory usage is measured only in the memory usage benchmark because it can negatively impact the execution times of the other benchmarks if it is incorporated within them. Furthermore, the size of the data structure does not change at all during the *static problem benchmark* and remains relatively constant throughout the *dynamic problem benchmark* as the number of *delete* and *insert* operations is equal. We could not find an uniform approach to measure memory usage that works for each of the languages, so we had to use language-specific tools that are described below.

Python. Measuring the memory consumption in Python was carried out with the help of Pympler[6], a package that provides functionality for, among other things, measuring object's size in memory. The package implements methods that calculate the size of a given object and using them is as easy as printing something in the command line interface.

Java. To measure the size of the data structure that is being tested in Java we have used the jcmd[7] tool to dump the contents of the heap, where the object is stored, into a file. We then used VisualVM[8] that provides a clear, human-readable interface to look into the snapshot of the heap. With VisualVM we can look at the memory usage per class and the number of objects of that class. Unfortunately for us, we couldn't see the size of the desired HashMap object but only its reference to the standard library class instance that is used by Java, namely *HashMap$Node*. The node objects represent

[4]https://scipy.org/
[5]https://numpy.org/
[6]https://pypi.org/project/Pympler/
[7]https://docs.oracle.com/en/java/javase/19/docs/specs/man/jcmd.html
[8]https://visualvm.github.io/

HashMap entries, and we were able to see the total size consumed by them but apparently Java was using objects of this type in other classes as and their number slightly exceeded the number of entries in our HashMap. That left us with a pretty good approximation which we could improve even further by manually calculating the size of our HashMap by multiplying the size of an entry, which we are able to see, times the number of entries.

C#. In C#, we measured the memory usage of the dictionary with the Diagnostic Tools that are integrated in Visual Studio 2022[9]. These tools allow you to take a snapshot of the heap and then look at the memory usage of objects per type. Since, Diagnostic Tools, allowed us to filter out any objects that are not in our code, finding the memory usage of our dictionary was a very simple task that did not require any manual calculations.

2.4.2 Execution times. Since we are measuring the performance of the Map implementations of three different languages, it is important to note that there are language specific differences that are independent of the implementations, but nevertheless can affect the execution times. Specifically, we are talking about differences in their compilers, or lack of such, and their runtime systems. It is arguably impossible to put all three languages on equal grounds due to the nature of these differences, however elaborating them here will provide important context for the final results of our experiments and will help the reader decide which language suits their needs the best. In each of the languages, we used some of the available options of their runtime environments in an attempt to deal with some of the differences and to make the benchmarks more consistent. Below you will find more detailed information for the case of each language and the tools that were used to measure their execution time in the benchmarks.

Python. The default implementation of Python is called CPython and its runtime environment does not provide a compiler that creates native machine code. Python code is only compiled to bytecode that is then being interpreted line by line by the interpreter of the runtime system. This puts Python in significant disadvantage in terms of execution speed as interpreted languages that are being interpreted have a quicker start-up than compiled (to machine code that is) but their execution speed is slower. The "-m compileall" option of CPython runtime environment compiles all Python code to bytecode ahead of time and then at runtime it only loads those files. This increases start-up times even further but does not influence execution times whatsoever.

To measure the execution times of the benchmarks in Python we used *time.time()* method that returns the current UNIX[10] time in seconds. For better readability and consistency with the other two languages the results were converted to milliseconds.

[9]https://visualstudio.microsoft.com/

[10]Unix time is a date and time representation widely used in computing. It measures time by the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970, the beginning of the Unix epoch

Java. Oracle Java Developer Kit (JDK)[11] provides a Java Virtual Machine (JVM) that is called HotSpot. The JVM is responsible for running Java code. In the case of HotSpot, this virtual machine comes with an interpreter and two bytecode-to-machine code compilers – Client and Server. The difference between the two is the trade-off that they have between compilation time and execution speed. The compilation times of the Client compiler are lower but the efficiency and the execution speed of the machine code that it produces is also lower in comparison to the Server compiler that has higher compilation times but significantly faster execution times of its machine code (technically, the Client compiler is divided into sub-levels with different balance between compilation times and execution speeds but that is not relevant to this experiment).

HotSpot implements something called Just-in-time compilation (JIT[12]). What that means is that by default when a Java program is ran its source code is first compiled to intermediate bytecode, just like Python (and C# for that matter as we are going to see later). This bytecode is then interpreted by the interpreter line by line. Methods that are executed a certain number of times are classified as warm and are being compiled by the Client compiler and afterwards the produced machine code for these methods is used instead of their bytecode. Methods that continue to be executed often after becoming warm become hot and are being compiled in the same way by the Server compiler.

The threshold that a method needs to reach to become warm or hot can be adjusted with few runtime system options and the Client compiler can be turned off completely the same way. For the sake of simplicity and consistency, we turned off the Client compiler for our benchmarks and reduced the threshold for methods to be compiled significantly. This reduced the preparation time that was necessary to ensure that the code of our benchmarks was compiled and not interpreted when measuring the performance of the data structure. We set a threshold for compilation of 10 and ran the benchmarks 300 times before measuring the performance to ensure that the compilation happens during this warm-up period and not when it mattered as this could have a significant negative impact on the performance.

We used *System.currentTimeMillis()* in Java that returns the UNIX time in milliseconds to measure the execution time of the benchmarks.

C#. The source code of C# in .NET[13] framework is first being compiled to Microsoft Intermediate Language (CIL) and then to machine code by the bytecode-to-machine code compiler of the framework at runtime. However, in contrast to Java, C# code is not interpreted at any point in time and instead each method is being compiled after it is first called. This could mean slower execution times the first time around a code is being executed in comparison to Python and Java, since the execution in essence freezes while waiting for the code to be compiled, while in Python and Java it is

[11]https://www.oracle.com/java/technologies/downloads/#java19

[12]https://en.wikipedia.org/wiki/Just-in-time_compilation

[13]https://dotnet.microsoft.com/en-us/

being interpreted without interruption.

DateTimeOffset.Now.ToUnixTimeMilliseconds() was used to get the UNIX time in milliseconds and to measure the execution time of the benchmarks.

Garbage collection. The runtime systems that we used for all three languages, namely – CPython 3.11, JDK 19 and .NET 7.0 – provided automatic garbage collection (GC[14]). The GC is a process on its own that requires resources to run and can be unpredictable as to when it takes place. This can impact the execution times that we measure and thus, can introduce inconsistencies. In the case of Python and C# This GC could be opted out easily but Java did not support such an option. This forced us to carry out the performance evaluation with active GC. Nevertheless, GC is an important and necessary part of the performance of a language, and in the real-world applications it is almost always a present factor, so evaluating performance in relation to GC can give important insights. To ensure some consistency in the results, each benchmark was run 10 times and the measured times of these runs were averaged out.

2.5 Statistical analysis

In the experiments conducted in this research, we compare the performance in terms of memory usage and execution times of three Map implementations, one from each language, by means of benchmarks. As we mentioned above, we have designed three benchmarks, one that evaluates the memory efficiency of the data structures and two that evaluate the performance by giving the data structures a specific, to the benchmark, problem to solve - either the static or dynamic membership problem. Furthermore, we have 10 different sizes of each problem, based on the setup set size $\boldsymbol{n}$, as described in the beginning of this section.

To make the results of our experiments statistically significant, the benchmarks for the static and dynamic membership problems are executed ten times for each problem size, to generate a sample of results that is supposed to negate the effect of any non-deterministic factors. Furthermore, we calculated 95%-confidence intervals (CI) for each data structure on each problem size $\boldsymbol{n}$. In the figures in Section 3, the shaded intervals around the curves represent the computed CIs.

3 PERFORMANCE OF MAP IMPLEMENTATIONS

In this section, we will go through the Map implementations that we have selected from each language, discuss the design of the experiments and conclude by analyzing the results.

3.1 Choice of implementations

Below, we will discuss the available map implementations in each of the languages and our pick for each of them for the purpose of this experiment. In general, we tried to choose implementations that are based on similar concepts and are more or less equivalent in terms of functionality. All three implementations that we ended up choosing are implemented as a hash table, support *isMember*, *delete* and *insert* operations and provide constant-time performance for these operations, assuming their hash function the elements properly among the buckets.

Python. In CPython 3.11.0 dict[15] data structure is the only mapping type. This implementation behaves like a hash table, and there are no alternative implementations of this data structure in CPython.

Java. JDK 19 provides various Java implementations for a variety of use cases. The most generic one is the HashMap[16] that is implemented as a hash table. There are sorted implementations such as TreeMap[17] and thread-safe ones such as the ConcurrentHashMap[18]. The Java Collections Framework (JCF) also provides the LinkedHashMap that guarantees the order of the *key-value* entries when the structure is being iterated over as the generic implementation does not have that property LinkedHashMap[19] . We chose to evaluate the HashMap implementation since no other implementation available in JCF provides better performance, even though some of them provide the same or slightly worse one in exchange for other advantages, like it is the case with the LinkedHashMap that we mentioned earlier. The HashMap implementation is also most similar to our choices for Python and C# and thus makes it the best option for direct comparison.

C#. The available dictionary implementations in .NET 7.0 are the generic dictionary, the sorted dictionary, the concurrent dictionary and their corresponding immutable variants[20]. However, the generic dictionary provides the best execution performance, and it is also the most similar to our selection for Python and Java, both in terms of functionality and implementation as it is also implemented as a hash table[21].

3.2 Experimental Design

The selected implementations discussed in the section above will be evaluated on solving the static and dynamic membership problem for setup sets of increasing size $\boldsymbol{n}$, where $\boldsymbol{n}$ is computed in the following way – $[n = (2^{20}/10) * k]$ – for k in the set $S = \{1, 2, 3..10\}$, meaning that we are going to have setup sets of 10 different sizes ranging from one $[10^5]$ to $[10^6]$. The *dynamic problem set* in each *dynamic problem benchmark* will have the same size as the setup set for that benchmark, while the *static problem set* will always have size of 2^20. The data in every set is uniformly distributed and the benchmarks conducted as explained in Section 2. To account for non-determinism such as GC mark and sweep cycles that can be detrimental for performance and to ensure that the benchmark results are statistically significant, each benchmark will be repeated at least 10 consecutive times and at the end the mean of all 10 samples

[14]https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

[15]https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects
[16]https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/HashMap.html
[17]https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/TreeMap.html
[18]https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/ConcurrentHashMap.
[19]https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/LinkedHashMap.html
[20]https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.idictionary-2?view=net-7.0
[21]https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-7.0

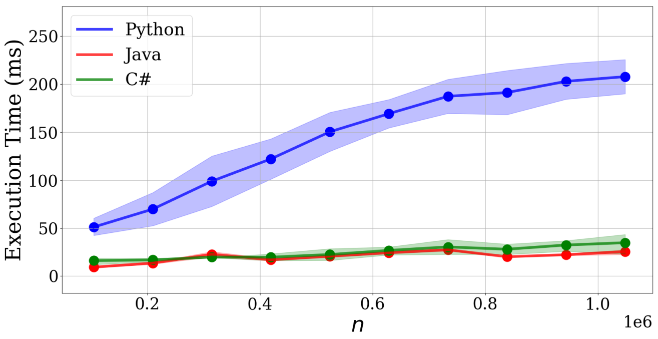

Fig. 1. Observed execution times of static membership problem plotted against ***n***

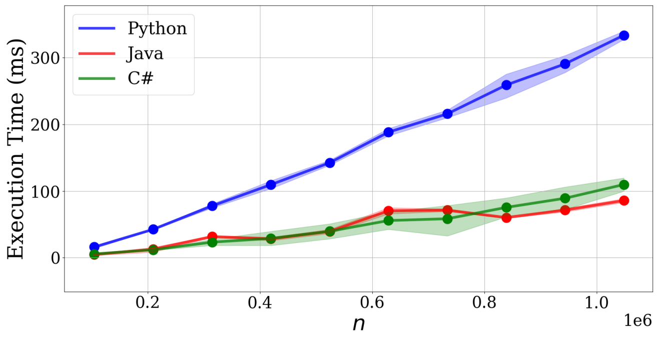

Fig. 2. Observed execution times of dynamic membership problem plotted against ***n***

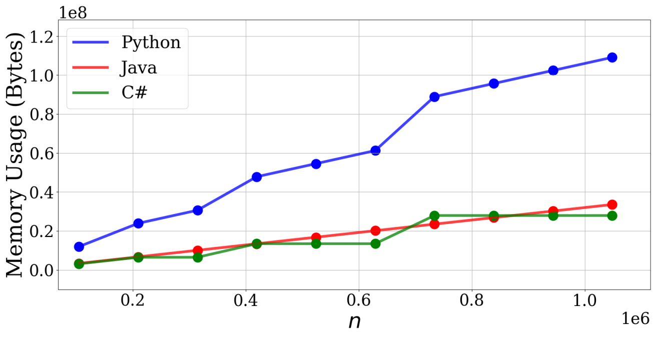

Fig. 3. Observed memory usage plotted against ***n***

will be used to represent the performance of the data structure on that specific benchmark.

3.3 Results, Discussion and Conclusion

After completing the experiments, we stored the results and processed them. The recorded sample means for each benchmark were plotted against the setup set size ***n***. Figure 1 shows the resulting graph for the static membership problem, the graph for the dynamic membership problem can be seen in Figure 2 and the results of the memory usage benchmark can be found in Figure 3. The graph in Figure 3 does not have shaded areas that represent the confidence interval as in Figure 1 and Figure 2 because the memory usage of each data structure that we measured remained unchanged regardless how many times we ran the memory benchmark and it was entirely dependent on the *setup set* size ***n***. Looking at the graphs for static and dynamic membership problems, we can see that for the lower values of ***n*** the execution times of all three data structures are not much different, but this difference quickly grows as ***n*** grows. In both graphs Python's curve appears to be as that of a linear function in relation to the curves for Java and C# that remain relatively flat. We can see that in both experiments, Java and C# performed very similarly with a slight edge for Java.

The graph for the memory usage is also not much different than the other two. Again, all three data structures have very similar memory usage for low values of ***n*** but that quickly changes with the increase of ***n***. Again, the difference between Java and C# is very small throughout the entire experiment and their curves have a small incline relatively to that of Python. Its curve again exhibits an incline close to that of a linear growth and as a result the gap between its curve and those of Java and C# grows as ***n*** increases. This time around, C# has a slight edge over Java's memory usage. Surprisingly, the curve of C#'s memory usage has few flat sections that indicate no growth in memory usage as the size of the data structure increases.

Definition 3.1. (Hash function.) A hash function maps the universe U of keys - $h : U \rightarrow \{0, \ldots, m-1\}$ to array indices or slots within the table for each $h(x) \in (0, \ldots, m-1)$ where $x \in U$

Since all three implementations of the Map data structure are implemented as hash tables[22] in theory they should have time complexity of $\theta(1)$ for the basic operations – *isMember*, *delete* and *insert* – and space complexity of $\theta(n)$ in the average case. In the worst case, the time and space complexity are defined by $O(n)$.

How well a hash table implementation will perform depends on how often it will have to deal with its worst-case scenario, that is, how often hash collisions occur and how efficient is the method for collision resolution. If the hash function used to hash the *keys* of the Map entries often creates the same hash for different *keys* then a lot of collisions will occur.

However, regardless of how good a hash function is, it is impossible to not produce any colliding hashes. Perfect hash function can be created only if the *keys* are known ahead of time[3]. The likelihood of collision increases as the size of the data structure grows, because hash functions produce a limited number of mappings, as shown in Definition 1, that start to repeat once the size of the universe U exceeds the number of unique hashes that the hash function can produce. Thus, having an efficient method for collision resolution is equally important for data structures of big size.

All three implementations that we tested use different hash functions (see Definition 3.1) based on the size of the Map object. In the beginning they use simple hash functions that can be computed

[22]https://en.wikipedia.org/wiki/Hash_table#Collision_resolution

easily but can produce unique hashes for very small universes U. As the Map object size increases more complex hashing functions are used instead that provide more table slots, also known as *"buckets"*, for the price of higher computation times.

When it comes to collision resolution, C# and Java uses the method of Separate chaining while Python implements the method of Open Addressing. The separate chaining method maintains a linked list[23] for every *"bucket"* and when there is a hash collision, the collided *keys* are chained together. Open addressing on the other hand, just looks for another empty *"bucket"* in the table in the case of hash collision. If the hashing function does not distribute the elements uniformly, that results in clusters which increases the performance time. In addition, the performance of Map implementations that use this method of collision resolution worsen as the table becomes more and more full[6]. This makes open addressing a potentially slower method for collision resolution than separate chaining. The results of our static problem experiment exhibit behavior that firmly fits this theoretical difference between the Java/C# implementations that we tested and that of Python. While the curves of Java and C# have very small inclines and remain relatively flat, the curve of Python grows linearly as the size of $\boldsymbol{n}$ increases. Similar result is observed in the graph for the dynamic problem. What is interesting to see is that Python's Dictionary also uses the most memory. In theory Java and C# should use more since they have to maintain linked lists for every *"bucket"* when there is a collision. Perhaps there are other language differences, besides the Map data structure implementations, between Python and Java/C# that contributes to this difference. It is also important to keep in mind that there is fundamental difference between the execution of Python's code and Java/C#'s one in the runtime systems that we tested as explained in Section 2.

4 CONCLUSION

In Section 3, we saw that Java's HashMap and C#'s Dictionary have very similar performance in terms of execution time and memory usage, with slight edge for Java in the former and one for C# in the case of the latter. We also have shown that Python performed significantly worse in both regards in our experiments. While the three implementations that we have compared in this research are based on hash table, they Python uses different method for collision resolution that may perform worse than the one used by Java and C#'s implementations, as discussed at the end of Section 3. However, the results of the memory usage experiments deviated from the expected outcome that the difference in collision resolution method suggested, with Python again performing the worst. In Section 2 we noted that there are differences in the runtime systems that we used for each language, that puts Python in disadvantage by default, nevertheless this research aimed to evaluate and compare the performance of CPython's Dictionary, JDK's HashMap and .NET's Dictionary regardless of their differences.

REFERENCES

[1] P. Drake. 2006. *Data Structures and Algorithms in Java.* Pearson/Prentice Hall.
[2] M. Hort. 2007. A survey of performance optimization for mobile applications. *Institute of Electrical and Electronics Engineers Inc* 48, 8 (Jan. 2007), 2879–2904. https://doi.org/10.1109/TSE.2021.3071193
[3] Balaji; Bonomi Flavio Lu, Yi; Prabhakar. 2006. Perfect Hashing for Network Applications. *IEEE International Symposium on Information Theory* (2006). https://doi.org/10.1109/ISIT.2006.261567
[4] Leonardo Pasquarelli. 2022. Extending Java Collections for List and Set Data Structures. http://essay.utwente.nl/91726/
[5] R. Saborido. 2018. Getting the most from map data structures in Android. *Empirical Software Engineering* (2018).
[6] & Mehlhorn K. Sanders, P. 2008. *Algorithms and data structures.* Springer.
[7] M. Voorberg. 2021. A performance analysis of membership datastructures in Java. http://essay.utwente.nl/87064/
[8] & Sedgewick R. Wayne, K. 2011. *Algorithms.* Addison-Wesley.

[23] https://en.wikipedia.org/wiki/Linked_list