Extending Java Collections for List and Set Data Structures

LEONARDO PASQUARELLI, University of Twente, The Netherlands

Java is a widely used programming language, running on different devices and used by many businesses. The Java Collections Framework is the official backbone of data structure and algorithm related matters. Nowadays, there are other Collections frameworks, providing data structures. However, such frameworks can provide a much better performance than the Java Collections. This raises the question of how the Java Collections can be modified or extended to achieve greater performance. We investigated how this could be achieved for data structures that base their implementations on the List and the Set data structure. Our benchmarks showed that Unrolled Linked Lists and Skip Lists provide a faster List data structure, and that Bloom Filters provide a much lower memory consumption than sets.

Additional Key Words and Phrases: Java Collections, Performance Benchmark, Data Structure, List, Set, Filter

1 INTRODUCTION

There are many data structures and implementations thereof, provided by languages, libraries and frameworks. For most data structures within a certain category, the operations are the same, often times specified by an interface, as it is the case within the *Java Collections*.

We can distinguish between different data structures through some attributes: whether the order matters, whether duplicate elements are allowed, how elements are accessed, or whether elements are mapped to one another. The way that such data structures are implemented is mostly hidden to the developer.

The *Java Collections* is a framework that has been introduced several years ago and over time they were maintained and extended, adding support for modern functionality such as *Lambda Expressions* and *Streams* [3]. Other frameworks, such as *Guava* by *Google* [1] or the *Apache Commons Collections* [2] aim to offer similar functionality.

The amount of data structures provided by the Java Collections is rather small. For instance, the only implementations of the `List` interface are `LinkedList` and `ArrayList` [7] as shown in Table 1. An empirical study conducted by Costa et. al has shown that for most scenarios when using the Java Collections, there was an alternative implementation, that would have led to an increase in performance and reduced memory usage [16].

In this research paper, we try to identify the main set of operations for different types of data structures, re-implement variants from inside Java Collections and new data structures outside the Java Collections and evaluate the performance, eventually discussing if it would make sense to introduce the benchmarked data structures to the Java Collections.

RQ1: What list implementations can be introduced to the Java Collections, that provide better performance or memory usage over existing list implementations, for some use cases?
RQ2: What set implementations can be introduced to the Java Collections, that provide better performance or memory usage over existing list implementations, for some use cases?

Related work: Benchmarking the Java Collections [16, 17, 20, 25], Benchmarking techniques in Java [19, 21], Lists implementations [22, 23, 27–29] and Bloom Filters [13, 15, 18, 23].
Voorberg carried out a performance analysis of integer-based membership data structures in Java [34]. This includes data structures that support `insert`, `delete` and `isMember` operations. Additionally, he proposed a new hash map data structure, and he looked into ways to improve collision resolution in hashing-based data structures. Parts of the benchmarking setup are taken and adopted from his research.

The paper is arranged as follows: Section 2 will introduce details of Java Collections about its structure, design goals and related research work. Section 3 will outline the setup of the benchmarks, conducted in this paper. Sections 4 and 5 will analyse the implemented and other existing data structures for *Lists* and *Unordered Sets*, respectively. Eventually, Section 6 will summarize our research work.

2 JAVA COLLECTIONS

The *Java Collections* is the prominent framework, consisting of Interfaces, general-purpose, special-purpose and concurrent implementations thereof, and other classes.

Most implementations are part of one out of four families: the List, the Set, the Map and the Queue family. Lists are an ordered data structure that can be indexed and where duplicate elements are allowed. Sets can be either ordered or unordered, and they do not allow duplicate elements. Maps can be thought of as an extension to sets, as they map a key, to a value. Traditionally, sets and maps use hashing functions, to find a location in memory for each value. Finally, queues have a FIFO or, in the case of a Stack, LIFO ordering. They do not support indexing, but they allow for duplicate elements.

For each of the families, there can be multiple interfaces, for instance `Set` and `SortedSet`. The list of all interfaces and implementations can be found in the Java Documentation [4].

[1]https://guava.dev/
[2]https://commons.apache.org/proper/commons-collections/

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

TScIT 37, July 8, 2022, Enschede, The Netherlands

Table 1. Overview of existing general-purpose implementations in the Java Collections [3]

Interface	Hash Table (HT)	Resizable Array	Balanced Tree	Linked List (LL)	HT + LL
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Dequeue`		`ArrayDequeue`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

Next to the general-purpose implementations shown in table 1, there are both special-purpose and concurrent implementations for different uses. Interestingly, not all concurrent implementations have a non-concurrent implementation. By way of example, there is a concurrent implementation of the Skip List Set named `ConcurrentSkipList` [5], but there is no non-concurrent implementation of the Skip List Set. This suggests that some concurrent data structures could be re-implemented without support for concurrency, in order to increase performance when concurrent features are not needed.

The primary reason to use the Java Collections are included but not limited to a reduced programming and learning effort to the user, as well as an increase in performance [4]. The data structures implemented by the Java Collections often times have thousands of line and complex optimizations, to fulfil the latter goal.

As mentioned in the Introduction and as seen in Table 1, the quantity of general-purpose implementations for the shown interfaces is fairly small, with 3 implementations at maximum. Some of the cells are left empty, as it would not make sense to have an implementation of an interface with all of the listed data structures, simply due to redundancy. As an example, the only general-purpose classes implementing the `List` interface are `ArrayList` and `LinkedList`, whereas efficient general-purpose implementations such as *Unrolled Linked Lists* [32] and *Skip Lists* [28] are not implemented.

Famous competing existing libraries are the *Apache Commons Collections*, *Guava*, the *Eclipse Collections* (formerly known as GS or Goldmansachs Collections) and *Koloboke*, some of which were selected for an empirical study conducted by Costa et. al [16].

As found out in the same study, it is not trivial to select the correct Java Collection and implementations thereof: in contrary, a selection of unsuitable collections can lead to runtime bloat and higher memory usage [16]. At the same time, it was discovered that for most scenarios of the study, there was an alternative implementation that would have led to an increase in performance and reduced memory usage [16].

A metric that can be taken into consideration in addition to efficiency and memory usage is energy efficiency. This has become increasingly important in mobile devices, such as phones and laptops. Some of the most widely-used data structures from the Java Collections, including the `ArrayList` and the `HashMap` have bad energy-efficiency [25]. Energy efficiency will not be taken into consideration within this research paper, but it is a parameter which could be taken into account for future research work on the modification of the Java Collections.

[5]https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListSet.html

2.1 Choice of Data Structure Families

We chose to focus on the `List` and `Set` interfaces. On one hand, those two interfaces have wide use cases, and we assume that they are amongst the most used interfaces. On the other hand, there exist a number of implementations not present in the Java Collections or improvements to be taken on existing implementations, as shown in Chapter 5.

All data structures are implemented either by ourselves, or they are adapted from the Open Data Structures book [23] or other licensed repositories. We do not implement all methods specified by the interface, but the ones for which the performance is relevant.

Initially, we implemented the data structures by ourselves. However, this approach was both time-consuming and error-prone, and we soon realized that we gain little from implementing complex classes by ourselves. Therefore, for some data structures, we reused and adapted methods or classes from licensed open source repositories or from books [23].

The data structures were found out about by reading books about algorithms and data structures, or by visiting programming threads. At the start of the research, our focus was on improving hashing collisions, by using *Cuckoo Hashing*, and comparing tree implementations against each other, such as *Scapegoat Trees*, *Splay Trees* and *Left-Leaning-RBTrees.* After a while, our focus shifted to *Bloom Filters*, *Unrolled Linked Lists* and *Skip Lists*, since our impression was that these data structures would have a larger use-case over the previously mentioned ones.

3 METHODOLOGY

For all the categories that were tested, benchmarks were performed, making benchmarks an essential part of this research. Due to the similarity of the research topic, many benchmarking methods were adopted from Voorberg's paper in 2021 [34].

3.1 Benchmarking Structure

A benchmark can be split into two phases: the setup phase and the experiment phase. In the setup phase, a varying number of elements is inserted into the data structure, in order to create a base layer for the experiment. In the experiment phase, the methods of the data structures are executed in different combinations. For both experiments, the sum of the actions are measured. For some experiments, only a subset of the operations will be measured, in order to get a more accurate result.

The number of instructions in the experiment phase is always, 2^{20} and the number is shared equally throughout the instruction types. This number was used by Voorberg's research to cover a large variety of performance cases, and it was equally optimal in our benchmarks [34]. The setup phase inserts 2^n elements into the list, where $n = \{10, 11, .., 20\}$. Again, we found that this range gave us representative results, while keeping the run time at a bearable level. The benchmarks are always applied to uniformly and normally distributed data.

3.2 Benchmarking in Java

Benchmarking in Java is quite different than in other languages, due to factors such as Garbage Collection (GC), the Java Virtual Machine (JVM) and Just-In-Time compilation (JIT). There are a number of different ways to perform benchmarks, by for instance running a test over multiple iterations over the same data set or different sets, only taking the best runs or by excluding compilation time, as found out in a research article from 2007 [19].

Throughout this research, we will take the median of 10 iterations over the same data set. This is done for a different number of elements, inserted in the setup phase. There will only be one VM invocation, such that we will measure the steady-state performance, which should resemble a real use case scenario as close as possible [19]. The same applies to GC: similarly to Voorberg's research work [34], we are not aiming to reduce GC. Thus, there are some non-deterministic influences by the GC. However, this is important to our benchmarking, as we want it to be similar to a real use case scenario.

As suggested by researchers in 2015 [21], it is advisable to 'warm-up' the JVM, in order to reduce other non-deterministic influences, including JIT compilation time. Therefore, we will run the methods of the setup and experiment operations, before performing the actual benchmark.

3.3 Measurements

The performance is measured through the `System.currentTime` method in Java [11]. For each benchmark iteration, there will be a time measurement for the setup and for the experiment phase.

The memory usage is measured through the `jcmd` tool [5]. To our surprise, this was a lot harder than expected, because `jcmd` would group the memory usage for classes from the Java Collections, and it would only show the memory consumption for the class headers and the pointer to the internal classes.

We have tried using different commands in the `jcmd` utility other than `GC.class_histogram` and we tried to use *VisualVM*, but we faced the same problems there.

The most promising alternative to `jcmd` was `jol` [9]. Using `jol`, the memory values would be consistently shown for all data structures. However, the values were not reliable. When benchmarking the Bloom Filter and the Counting Bloom Filter, the Bloom Filter had the same memory consumption as the Counting Bloom Filter, although the memory consumption of the Counting Bloom Filter should have been higher by a factor of 4. We faced the same problem, with the Array List and the Unrolled Linked List.

After talking to Voorberg, who used `jcmd` and faced similar issues, we decided to stick with theoretical computations for the exceptional classes. For data structures such as the *Bloom Filter* this was not an issue, since the its memory consumption is straightforward to compute. For classes such as the *Unrolled Linked List*, it was much harder to compute, since the data structure could resize. We ended up inspecting the class instances in the debugger of the benchmarks, and basing our computations on that data, as it would help us approximate the real memory consumption value.

Both values are measured for each iteration of each file, distinguishing between setup and experimental time. Whenever applicable, only a subset of operations will be measured, in order to get a better idea of how long different operations take.
The testing environment was running Arch Linux 5.16 (64bit) with an i7-4700MQ processor and 16GB of RAM.

3.4 Statistical Analysis

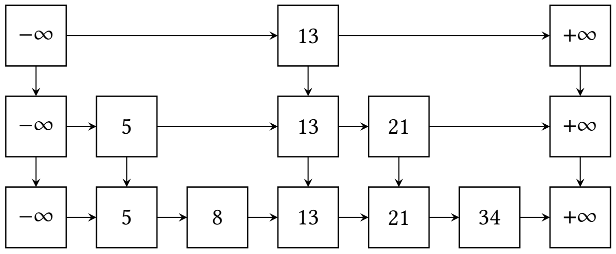
In our benchmarks, we have 10 iterations for each value of n, in which three to four data structures are compared. We compute 95% Confidence Intervals, providing us with an interval that the values will appear with a probability of 95% [30, p. 131]. In order to compute the confidence interval, the distribution of the data needs to be known. We conduct a KS test, to check the distribution that our result data adheres the most to [30, p. 175]. Our results have shown, that it mostly follows an *inverse gamma* distribution for timed experiments, and a normal distribution for memory experiments. The prior is the same as in Voorberg's research, which is likely to happen because of the similar benchmark setup. Thus, the experiment data is fit into the corresponding distribution, and afterwards the CI intervals are computed. In the plots, they are represented by the shaded areas surrounding the mean line.

4 LISTS

List data structures have two important properties: elements are ordered, and duplicate elements are allowed. Therefore, elements can be selected by their index, which is less common in other data structures. Lists typically support the following operations [23, p. 7]:

- `size()`: returns the length of the list
- `get(i)`: returns the value of x_i
- `set(i, x)`: sets the value of x_i equal to x
- `add(i, x)`: inserts x at index i
- `remove(x)`: removes the element x_i

Fig. 1. Skip List of height 3 with 5 elements.



These methods rely on finding an element by its value and optionally doing an operation (`contains`, depending on the implementation also `remove`), or accessing an index and doing an operation (`get`, `set`, `add`).

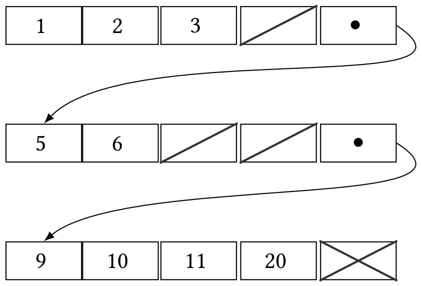
Although lists are *ordered*, it is important to note this property does not imply that lists are *sorted*. In unsorted lists, operations such as element querying can have different look up times, depending on whether the list is sorted or unsorted.

Provided implementations of the `List` interface are `ArrayList` and `LinkedList`, which is actually a *Doubly Linked List* [6], as well as synchronized Lists [7], which we are not relevant in this research work. Array Lists support direct lookup, which ensures that `get` and `set` operations are performed at $O(1)$. For `add` and `remove`, elements of the array may need to be shifted, which gives a worst-case run time of $O(n)$. Because Doubly Linked Lists do not support direct look, but instead elements are found by iterating over the nodes, the runtime for `get` and `set` is $O(1+\min(i, n-1))$. As adding and removing elements only requires to adjust links, the runtime for those operations are the same as for `get` and `set`.

4.1 Modern implementations

Some of the list implementations that have not made it into the Java libraries yet are *Unrolled Linked Lists* and *Skip Lists* [28, 29, 32], both of which have been part of extensive research on variants, such as *Concurrent Unrolled Linked Lists*, or *Deterministic Skip Lists* [24], or theoretical analyses on Skip Lists [26, 31].

Fig. 2. Illustration of an Unrolled Linked List with 3 blocks of size of 4 and 9 elements



A further improvement to Doubly Linked Lists can be made by storing the XOR operation of the neighbouring node addresses, instead of the two addresses [33]. However, due to the lack of using pointers in Java code, such a data structure cannot be implemented.

Unrolled Linked Lists can be seen as Linked Lists consisting of arrays, called blocks, as shown in Figure 2. Not only does this make indexing faster, but there is also less overhead due to a single pointer needed for multiple elements, instead of just a single element [23, p. 71]. Whereas Linked Lists have the disadvantage that their data is spread out in memory, and Array Lists have the disadvantage that they insert and delete at $O(n)$ due to shifting operations, Unrolled Linked Lists minimize both problems [22].
The conventional Skip List data structure is probabilistic, as some randomness is involved when inserting elements. Due to its run time, it is often being compared to balanced trees. Skip Lists can be thought of Linked Lists, with multiple links to next elements, setup similarly to binary trees. Skip Lists provide a similar run time to balanced trees, which makes them highly efficient [28]. While the worst case run time is better for balanced trees than for skip lists [28], it has been shown that the worst case run time is achieved rarely in practice [31]. An overview of the run times of the list implementations can be found in Table 2.

4.2 Experimental Design

The data structures used in this experiment are the *Array list*, *Doubly linked list*, *Unrolled linked list* and the *Skip list* The data consists of a generated list of indexes that are within the interval $[0, \texttt{list.length}]$. We split the experiment into an experiment with `get` and `set` instructions, and an experiment with `add` and `delete` instructions, since those two pairs of methods have similar operations and run times. In the `add` and `delete` test suite, the instructions alternate to ensure that the generated indices are in range of the list size. For the other test suite, the order of the instructions is randomized.

The split will also give us an idea of which list would be best suited, when it needs to be modified frequently, when it needs to be accessed frequently or when both cases are needed often.

We perform a total of 2^{20} instructions on lists of size 2^{10}, 2^{11}, ... 2^{20}. Finally, we decided to generate test suites for both uniformly distributed data and normally distributed data with $\mu = 2^{\mathsf{exp}-1}$ and $\sigma = 2^{\mathsf{exp}-3}$.

4.3 Benchmarking Results

4.3.1 Time behaviour. Overall, the results seem to align with the theoretical run times of each data structure. With its linear run time, the *Linked list* has by far the worst performance, and we couldn't include it in most experiments as the benchmarks would take several minutes to finish.

For the `get` and `set` benchmark, the *Array list* with stellar $O(1)$ run times outperformed the the other two data structures. However, both the *Unrolled linked list* and the *Skip list* still had decent performance,

Table 2. Runtime of List implementations for n elements, operations at index i and $b = \sqrt{n}$ blocks [23, p. 23]

List	`get(i)/set(i,x)`	`add(i,x)/remove(i)`
ArrayList	$O(1)$	$O(n)$
DoublyLinkedList	$O(1 + \min\{i, n-i\})$	$O(1 + \min\{i, n-i\})$
UnrolledLinkedList	$O(1 + \min\{i, n-i\}/b)$	$O(b + \min\{i, n-i\}/b)^A$
SkipList	$O(\log n)^E$	$O(\log n)^E$

^{A} - Amortized run time
^{E} - Expected run time

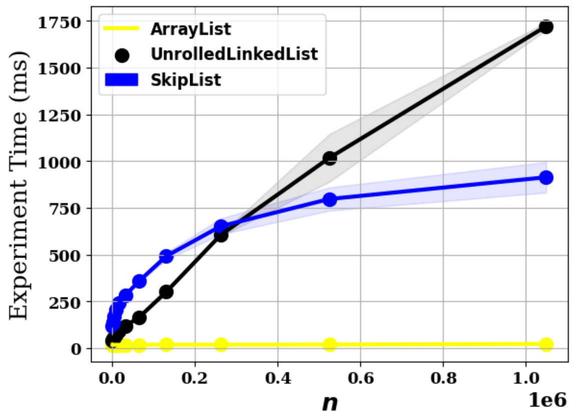


Fig. 3. Performance of get and set methods benchmarked on uniform data

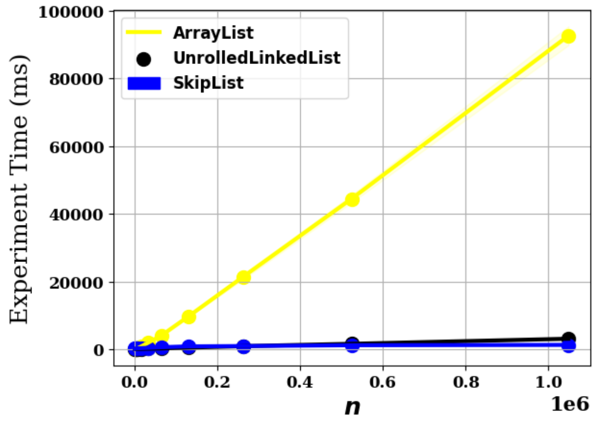


Fig. 4. Performance of `add` and `remove` methods benchmarked on uniform data

with the *Skip List* performing better for an increasing amount of elements, due to its logarithmic run time. The result for the `get` and `set` benchmark can be seen in Figure 3.
In the `add` and `remove` benchmark, the *Unrolled linked list* and the *Skip list* had about the same performance as in the previous benchmark suite, since the theoretical run times are about the same. The *Array list* on the other hand had an extremely high run time, due to the need of shifting elements in the array after each `add` and `remove` instruction. The result can be seen in Figure 4.
We got similar results for normally and uniformly distributed data for all benchmarks.

4.3.2 Memory behaviour. We had to manually compute the memory consumption of the Unrolled Linked List. The Array List had the lowest memory consumption, which we expected.

The Skip List had a similarly high memory consumption as the Linked List. Although that is very high memory consumption, the Skip List proves to be a much faster list implementation than the Linked List, at the same memory consumption.

Our computations of the memory consumption of the Unrolled Linked List have shown that it is slightly worse than the memory consumption of the Array List, but better than the memory consumption of the other two data structures. We expected this result,

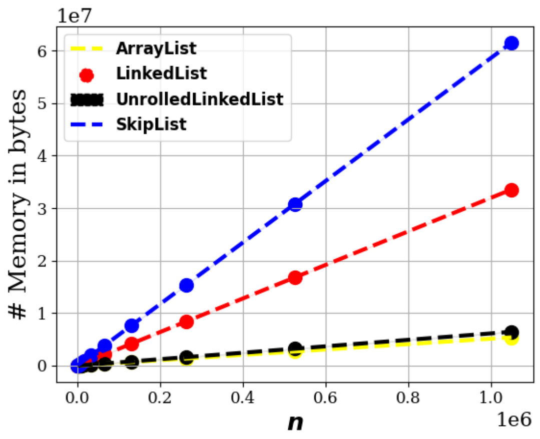


Fig. 5. Memory consumption for accessing lists (uniform data)

since the Unrolled Linked List can be seen as a combination of arrays and a Linked List. In practice, it is likely that the memory consumption of the Unrolled Linked List is slightly higher than in our computations, due to some overhead that was not visible to us. The memory usage can be seen in 5.

4.3.3 Conclusion. We have seen an incredible improvement in the performance of `add` and `remove` instructions for the new list implementations, at the cost of a higher memory consumption and slower direct access times, than with Array Lists. In use cases where an ordered and indexable data structure with frequent modifications is needed, the two implementations proved their value in the experiments. In any case, the two data structures are an improvement over the Linked List.

Hence, we think that it would make sense to introduce one, if not both of the data structures to the Java Collections. Our results have shown that both the Skip List and the Unrolled Linked List provide their own benefits: while the Skip List has a logarithmic and thus more scalable run time for `add` and `remove` instructions, the Unrolled Linked List has a lower memory consumption.

5 UNORDERED SETS

Unordered sets are unordered and duplicate elements are not allowed. Many set implementations make use of hash functions, due to the elements being unique. This results in direct lookup and thus very fast run times.

The operations supported by Sets are the following [23, p. 8]:

- `size()`: returns the size of the set
- `add(x)`: adds x to the set, if not already present
- `remove(x)`: removes x from the set, if present
- `contains(x)`: checks whether x is a member of the set

Although the theoretical worst-case run time is $O(n)$, in practice, if the table size is large enough, i.e. if the load factor is below 1, meaning that it is expected that there is just one element per bucket, one can expect a run time of $O(1)$.

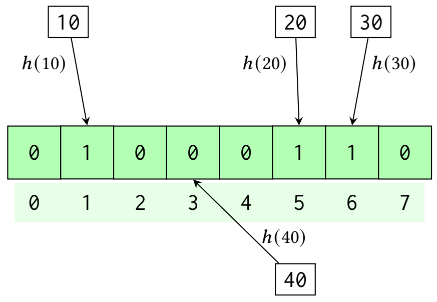
The only general-purpose unordered set that is available for use in the Java Collections is the `HashSet` [10]. The `HashSet` implementation in Java relies on the Java `HashMap` implementation, which uses self-balancing Trees as a chaining method for collision resolutions.

Looking back at Section 4, if one were to implement a Hash Set or Hash Map by themselves, a *Skip List* could be used as a chaining method for collision resolution. We do not think that it would be advisable to introduce such a data structure to the Java Collections, because the advantages of a *Skip List* cannot be applied to this case: it does not matter that *Skip Lists* are easier to implement than self-balancing trees, because the implementation is not a concern of the user of the Java Collections, but of the developers of the Java Collections, which does not provide value to the user. On top of that, *Skip Lists* do not have a better run time than self-balancing trees.

5.1 Modern Implementations

Besides the Hash Set, famous unordered sets are Bloom Filter implementations. *Bloom Filters* and variations thereof are one of the most time and memory-efficient set implementations that one could use [13, 18].

Fig. 6. Illustration of a Bloom Filter using a single hash function together with the values 10, 20, 30 and 40. Elements with the same hash values as 10, 20 and 30 have been inserted in the Bloom Filter before. No elements with the same hash value as 40 have been inserted, which means that 40 is not present in the Bloom Filter.



Bloom filters update the table, without storing the value of an element or its hash. For conventional Bloom Filters, this means that adding an element to the Bloom Filter will set the bit of some hashed values to 1. An example of a Bloom Filter using a single hash function is depicted in 6.

Although much more efficient than Hash Maps, this introduces the problem of False Positives. In other words, the result of the `contains` query is either *False* or *Possibly True*. On top of it, element removal is no longer possible, as it would interfere with other elements with the same hash code.

Another problem is that table resizing is not possible, because the values of the elements themselves are not stored. This means that one would carefully have to choose the table size before creating bloom filters.

One way to add element removal is by using *Counting Bloom Filters* [18], a variation, where the single bit is replaced by a numeric counter. When running any instruction, the element will be hashed multiple times, and the instruction is performed upon all hash value indexes.

To resolve other issues such as the resizing issue, other variants including the *Scalable* Bloom Filters have been proposed [12, 35].

The run time of the Bloom Filters is $O(k)$ regardless of the amount of elements present in the filter, where k denotes the amount of keys. In comparison, for the Hash Set with Linked List chaining, the run time varies on the amount of elements, where it can take between $O(1)$, and in the unlikely worst case, O(n).
The memory usage is easy to compute: Bloom filters require $-1.44 \log_2 \epsilon$ bits per element [15]. At a false positive rate of $\epsilon = 0.01$, this results in 9.6 bits per element. Our implementation of the Counting Bloom Filters takes 4 bits of space per element, meaning that the memory usage is at roughly 38 bits per element.

Table 3. Time complexity of set data structures for n elements and k hash functions

Set	`add(i)`	`remove(i)`	`contains(i)`
BloomFilter	$O(k)$		$O(k)$
CountingBF	$O(k)$	$O(k)$	$O(k)$
HashSetLL	$O(n)$	$O(n)$	$O(n)$
HashSetLL^E	$O(1)$	$O(1)$	$O(1)$
HashSetRBTree	$O(\log n)$	$O(\log n)$	$O(\log n)$

5.2 Experimental Design

The data structures used in this experiment are the *Hash set* with *Linked list*-chaining, *Hash set* with *Red Black Tree*-chaining, *Bloom filter* and *Counting Bloom filter* We generate a list of uniformly and normally distributed values that are added, removed or queried. We decided to add the *Red Black Tree* to have another data structure for to benchmark the filters against.

Because conventional Bloom filters lack the ability to resize, we decided that the two Hash Maps would not need to resize either and that they would instead have a fixed size, which would balance out the memory usage and the run time.

We have multiple test suites to isolate the different methods: a `query` test suite a dynamic test suite with `add`, `query` and `delete` instructions and finally a test suite with `add` and `query` instructions. Normal bloom filters do not support element removal. Therefore, they are not included in the second test suite. Since we still wanted to benchmark their `add` performance, we added the last test suite.

The split will give us an idea of which data structures would be suited when the set needs to be modified frequently, when it needs to be accessed frequently, or when both are needed.

We perform a total of 2^{20} instructions on sets of size 2^{10}, 2^{11}, ..., 2^{20}.

All of our Set data structures use the Murmur3 hashing function, introduced by Austin Appleby, to hash elements [1]. We chose Murmur3 as our hashing method, because we were looking for a fast and therefore non-cryptographic, but reliable hash function, which accepts a seed to support multiple hash iterations. Murmur3 satisfies our requirements. As a side note, it is already implemented in the Apache Java Collections [8].

Our Bloom filters have a false positive rate of 1%, which determines the size of the set. We concluded that this false positive rate would be low enough to be rendered as useful. The number of hash functions is derived from the size of the set.

5.3 Benchmarking Results

5.3.1 Time Behaviour. To our surprise, the performance of the Bloom Filters was not much better than the performance of the other data structures. For higher List sizes, the Bloom Filters were faster, whereas for smaller Set sizes, the Bloom Filters were noticeably slower. We expect that this has to do with the high amount of hashing that the Bloom Filters undergo. This makes the Bloom Filter slow at the start, but overall more scalable. The performance of the Counting Bloom Filter was very similar to the one of the Bloom Filter.

The *Hash set* with *Linked List* chaining was better for a low amount of elements, but worse for a large amount of elements. The performance of the *Hash set* with *Red Black Tree* chaining was bad when there were `add` instructions, due to re-balancing, but for querying, the performance had good scalability.

The performance of the *Hash set* with *Linked list* chaining was slightly worse when using normally distributed data, since there were more collisions and therefore longer chains of *Linked lists*.

The performance of the data structures can bee seen in Figures 7 and 8.

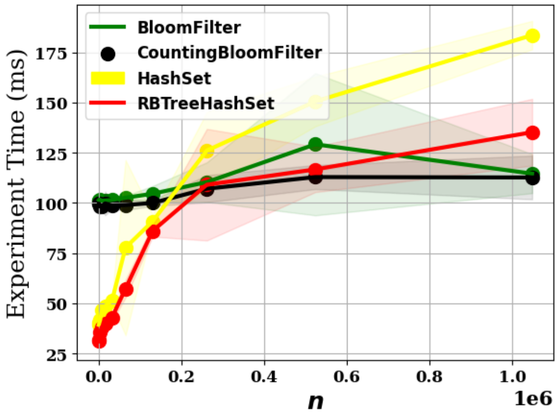


Fig. 7. Querying test for set data structures (normal data)

5.3.2 Memory Behaviour. We had to manually compute the memory consumption of the Bloom Filters. Therefore, we cannot fully rely on the results. Nevertheless, our results have shown that the Filters have a much lower memory consumption than the other data structures.

This is most likely because very few bytes are allocated for each expected element, whereas the other data structures allocate four bytes for every element, on top of overhead data from the chaining method.

5.3.3 Conclusion. If memory consumption is important and false positives can be sustained, then Bloom Filters provide a solid alternative to HashSets. The performance could be improved as suggested by existing research [14]. Resizing is not possible with the two variations shown here, however, one could use a different variant [12, 35]. If one wanted to add Bloom Filters to the Java Collections, these

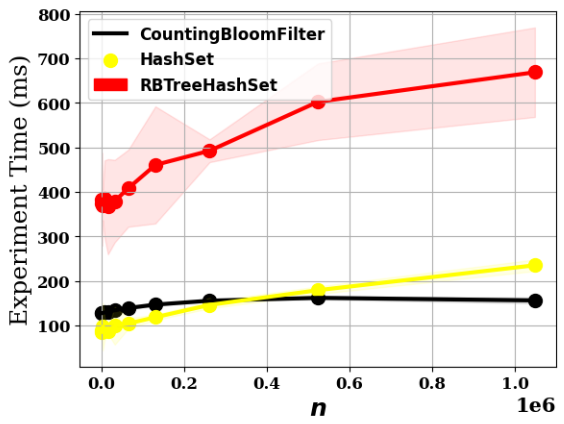


Fig. 8. Dynamic test case for Counting Bloom Filter and two Hash sets (normal data)

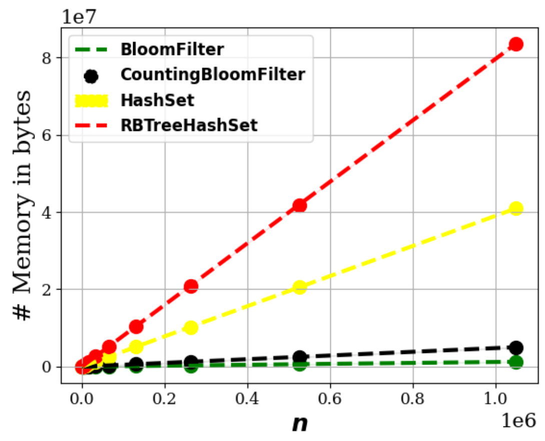


Fig. 9. Memory consumption for querying sets (normal data)

aspects would be need to kept into account.

Our benchmarks were performed with Integer values, requiring only 4 bytes of space. The Bloom Filter renders to be even more space efficient compared to the other data structures, in cases in which the data type is larger than an Integer. In that case, we still require very few bytes, compared to multiple bytes being required for storing the element alone.

Further investigation needs to be done, on which Bloom Filter variation would be the most fitting. Depending on the Bloom Filter that was chosen, some properties might clash with the `Set` interface defined in Java. Perhaps, it would make sense to introduce a `Bloom Filter` interface, in that case.

6 CONCLUSIONS

In this research paper, we investigated if faster or more memory-efficient List and Set data structures should be added to the Java Collections. Throughout the benchmarks, we have seen that for both the `List` and the `Set` interface, there are data structures that could be added in order to fulfil these requirements and thus offer a more versatile selection of data structures in the Java Collections. Both the List and the Set data structures implemented adhere to their theoretical run times in our benchmarks.

For **RQ1**, we have looked at the Skip List and at the Unrolled Linked List data structures. Both data structures heavily outperform the Linked List in terms of time, and they outperform the Array List for `add` and `remove` instructions. For scenarios in which lists with frequent modifications are necessary, they provide a great fit. Retrieving and replacing elements happens at linear run time with the Unrolled Linked List and with logarithmic run time for the Skip List, whereas it happens with constant run time for the Array List. However, their performance is still fine, when compared to the performance of the Array List when needing to add and remove many elements.

For **RQ2**, we have looked at the conventional Bloom Filter and at the Counting Bloom Filter. At the cost of False Positives, they provide a much more memory-efficient data structure, while providing scalable performance. Due to some differences to the `Set` interface, it could make sense to introduce a new interface.

An example where Bloom Filters could be useful is when developing a service which keeps track of millions of URLs. Instead of storing the entire URL string for each URL, the Bloom Filter only keeps track of whether the URL has occurred, leading to a decreased memory consumption. Google Chrome uses Bloom Filters to check if an entered URL is malicious [2]. Whenever the Bloom Filter returns a value of 0, we know for certain that the URL was not visited. Whenever the value is 1, the URL is checked in a different hash table to verify if it is malicious.

Other Bloom Filter variants need to be investigated, in order to seek out the best Bloom Filter variant for the Java Collections. It could also be investigated, if data structure implementations for other interfaces could be introduced to the Java Collections. As for modifying data structures, we could see from the research mentioned in Section 1 that implementations of other Collections can provide better performance for the same data structures. Therefore, some data structures of the Java Collections could be updated accordingly.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. Vadim Zaytsev for his extensive and early supervision and for supporting me with starting my Software Engineering internship in time and I would like to thank anyone who proofread my paper.

REFERENCES

[1] 2010. aappleby/smhasher. Retrieved June 25, 2022 from https://github.com/aappleby/smhasher
[2] 2012. Issue 10896048: Transition safe browsing from bloom filter to prefix set). Retrieved June 25, 2022 from https://chromiumcodereview.appspot.com/10896048/
[3] n.d. Collections Framework Enhancements in Java SE 8. Retrieved June 25, 2022 from https://docs.oracle.com/javase/8/docs/technotes/guides/collections/

changes8.html

[4] n.d.. Collections Framework Overview. Retrieved June 25, 2022 from https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

[5] n.d. The jcmd Utility. Retrieved June 25, 2022 from https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr006.html

[6] n.d. LinkedList (java Platform SE 8). Retrieved June 25, 2022 from https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

[7] n.d. List Implementations. Retrieved June 25, 2022 from https://docs.oracle.com/javase/tutorial/collections/implementations/list.html

[8] n.d. MurmurHash3 (Apache Commons Codec 1.15 API). Retrieved June 25, 2022 from https://commons.apache.org/proper/commons-codec/apidocs/org/apache/commons/codec/digest/MurmurHash3.html

[9] n.d. OpenJDK: jol. Retrieved June 25, 2022 from https://openjdk.java.net/projects/code-tools/jol/

[10] n.d. Set Implementations. Retrieved June 25, 2022 from https://docs.oracle.com/javase/tutorial/collections/implementations/set.html

[11] n.d. System (Java Platform SE 7). Retrieved June 25, 2022 from https://docs.oracle.com/javase/7/docs/api/java/lang/System.html

[12] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable bloom filters. *Inform. Process. Lett.* 101, 6 (2007), 255–261.

[13] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[14] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An improved construction for counting bloom filters. In *European Symposium on algorithms*. Springer, 684–695.

[15] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet mathematics* 1, 4 (2004), 485–509.

[16] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. 2017. Empirical study of usage and performance of java collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 389–400.

[17] Noric Couderc, Emma Söderberg, and Christoph Reichenbach. 2020. JBrainy: Micro-benchmarking Java Collections with Interference. In *Companion of the ACM/SPEC International Conference on Performance Engineering*. 42–45.

[18] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking* 8, 3 (2000), 281–293.

[19] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices* 42, 10 (2007), 57–76.

[20] Hans-Dieter A Hiep, Jinting Bian, Frank S de Boer, and Stijn de Gouw. 2020. History-based specification and verification of Java collections in KeY. In *International Conference on Integrated Formal Methods*. Springer, 199–217.

[21] Vojtěch Horkỳ, Peter Libič, Antonin Steinhauser, and Petr Tuuma. 2015. Dos and don'ts of conducting performance measurements in java. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. 337–340.

[22] Neha Jain. 2012. Optimization of Cache Memory Using Unrolled Linked List. (2012).

[23] Pat Morin. 2013. *Open Data Structures (in Java)*. Pat Morin.

[24] J Ian Munro, Thomas Papadakis, and Robert Sedgewick. 1992. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*. Citeseer, 367–375.

[25] Wellington Oliveira, Renato Oliveira, Fernando Castor, Gustavo Pinto, and João Paulo Fernandes. 2021. Improving energy-efficiency by recommending Java collections. *Empirical Software Engineering* 26, 3 (2021), 1–45.

[26] Thomas Papadakis. 1993. *Skip lists and probabilistic analysis of algorithms*. University of Waterloo Ph. D. Dissertation.

[27] Kenneth Platz, Neeraj Mittal, and Subbarayan Venkatesan. 2014. Practical concurrent unrolled linked lists using lazy synchronization. In *International Conference on Principles of Distributed Systems*. Springer, 388–403.

[28] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[29] William Pugh. 1998. *A skip list cookbook*. Technical Report.

[30] Dharmaraja Selvamuthu and Dipayan Das. 2018. *Introduction to statistical methods, design of experiments and statistical quality control*. Springer.

[31] Sandeep Sen. 1991. Some observations on skip-lists. *Inform. Process. Lett.* 39, 4 (1991), 173–176.

[32] Zhong Shao, John H Reppy, and Andrew W Appel. 1994. Unrolling lists. In *Proceedings of the 1994 ACM conference on LISP and functional programming*. 185–195.

[33] Prokash Sinha. 2005. A memory-efficient doubly linked list. *Linux Journal* 2005, 129 (2005), 10.

[34] Marten Voorberg. 2021. *A performance analysis of membership datastructures in Java*. B.S. thesis. University of Twente.

[35] Kun Xie, Yinghua Min, Dafang Zhang, Jigang Wen, and Gaogang Xie. 2007. A scalable bloom filter for membership queries. In *IEEE GLOBECOM 2007-IEEE Global Telecommunications Conference*. IEEE, 543–547.