Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Runtime Evaluation of Data Structures in Prolog

Bachelorarbeit

im Studiengang Informatik
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von
Christoph Ludolf

Beginn der Arbeit: 04. November 2022
Abgabe der Arbeit: 06. Februar 2023

Erstgutachter: Prof. Dr. Michael Leuschel
Zweitgutachter: Dr. J. Bendisposto

Selbstständigkeitserklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 06. Februar 2023

Christoph Ludolf

Abstract

Publicly available data on benchmarks of the (SICStus-)Prolog data structures Ordered Set, List (without duplicates), AVL Tree, Mutdict, Mutarray, Blackboard and Assert is sparse or does not exist. This thesis aims to fill this gab by providing benchmarks on the runtime and memory usage of performing the access/insertion operation on these data structures. Furthermore, it tries to answer questions about their performance. Mutable data structures are very fast, but how do they compare and which data structure performs best? Does the sorting of the Ordered Set provide an advantage over the other data structures when accessing elements that are not in the data structure? The impact of data structure size, number of operations and miss percentage on runtime is observed. In addition, the effects of the data structure size on memory usage are also observed. Data structure size describes the number of elements contained in a data structure. Number of operations describes the number of access/insertion operations performed per benchmark. Miss percentage describes a percentage of keys accessed that are not contained in a data structure when the access operation is performed. For each benchmark, one of the parameters varies while the other two are constant. Integer values, atoms and complex terms are used as keys. The results were obtained by creating datasets consisting of one of the key types and then performing the access/insertion operation on the data structures using the elements of the datasets. The results are visualized in the form of graphs. It is found that the mutable data structures Mutarray and Mutdict are very fast, closely followed by Assert, Blackboard and AVL Tree, and that Ordered Set and List are very slow. Only a small Ordered Set or List can outperform any of the other data structures. The Ordered Set proves to be worse than the List in most cases. The sorting of the Ordered Set offers only a small advantage, which depends on the key type and and the percentage of missed accesses. The results imply that the mutable data structure are preferable if runtime is important.

Acknowledgements

I want to thank my husband Nick Ludolf, my parents Ute Ludolf and Edgar Ludolf, my siblings Julia Ludolf and Martin Ludolf, as well as my thesis advisor Philipp Körner for their support. My family helped me a lot by freeing me from distractions and keeping me focused on the task at hand. My thesis advisor was always available to answer my questions and to offer advise.

Contents

1 Introduction **1**

2 Motivation **1**

3 Related Work **2**

4 Background **2**

4.1 Observed Data Structures . . . 3
4.2 Memory Parameter . . . 4

5 Procedure **5**

5.1 Random Number Generation . . . 5
5.2 Data Structures . . . 6
5.3 Data Patterns . . . 6
5.3.1 Values . . . 8
5.4 PC Specs . . . 8
5.5 Prolog Version . . . 8
5.6 Output . . . 8
5.7 Benchmark Access . . . 8
5.8 Benchmark Insert . . . 9

6 Results **9**

6.1 Access Operation . . . 9
6.1.1 Data Structure Size: Integer as Keys . . . 10
6.1.2 Data Structure Size: Atoms as Keys . . . 11
6.1.3 Data Structure Size: Complex Terms as Keys . . . 12
6.1.4 Miss Percentage: Integer as Keys . . . 12
6.1.5 Miss Percentage: Atoms as Keys . . . 13
6.1.6 Miss Percentage: Complex Terms as Keys . . . 15
6.1.7 Operation Count: Integer as Keys . . . 17
6.1.8 Operation Count: Atoms as Keys . . . 17
6.1.9 Operation Count: Complex Terms as Keys . . . 18
6.2 Insertion Operation . . . 19

6.2.1 Data Structure Size: Integer as Keys . . . 19
6.2.2 Data Structure Size: Atoms as Keys . . . 20
6.2.3 Data Structure Size: Complex Terms as Keys . . . 20
6.2.4 Operation Count: Integer as Keys . . . 22
6.2.5 Operation Count: Atoms as Keys . . . 23
6.2.6 Operation Count: Complex Terms as Keys . . . 23
6.3 Memory . . . 24
6.3.1 Global Stack Bytes Used: Access Operation with Integer Values as Keys . . . 24
6.3.2 Global Stack Bytes Used: Insertion Operation with Integer Values as Keys . . . 24
6.3.3 Global Stack Bytes Used: Access Operation with Atoms as Keys . . . 24
6.3.4 Global Stack Bytes Used: Insertion Operation with Atoms as Keys . . . 25
6.3.5 Global Stack Bytes Used: Access Operation with Complex Terms as Keys . . . 25
6.3.6 Global Stack Bytes Used: Insertion Operation with Complex Terms as Keys . . . 25
6.3.7 Heap Bytes Used . . . 25
6.3.8 Garbage Collector Calls: Access Operation with Integer Values as Keys 26
6.3.9 Garbage Collector Calls: Insertion Operation with Integer Values as Keys . . . 26
6.3.10 Garbage Collector Calls: Access Operation with Atoms as Keys . . . 27
6.3.11 Garbage Collector Calls: Insert Operation with Atoms as Keys . . . 27
6.3.12 Garbage Collector Calls: Access Operation with Complex Terms as Keys . . . 27
6.3.13 Garbage Collector Calls: Insert Operation with Complex Terms as Keys . . . 28
6.4 Other . . . 28
6.4.1 Bulk Access . . . 28
6.4.2 Loop Overhead . . . 28
6.4.3 AVL with fetch check . . . 29
6.4.4 Different Random Number . . . 29
6.4.5 Garbage Collection Time of Ordered Set . . . 29

7 Conclusions **29**

Appendix A Graphs **31**

List of Figures **45**

List of Tables **47**

List of Algorithms **47**

List of Listings **47**

References **48**

1 Introduction

In (SICStus[1]-)Prolog the commonly used data structures are List, Ordered Set, AVL Tree, Mutdict, Mutarray, Blackboard and Assert. The main point of this thesis is to answer the question which of these data structures performs best and how they compare. This is achieved by analyzing and comparing the runtime of the different data structures, which are described more in detail in Section 4.1. The runtime of the observed data structures is theoretically known, but there is no publicly available data on benchmarks of the runtime. It is tempting to point at the theoretic analysis of the runtime of data structures to avoid the task of benchmarking program code, but a practical observation offers new knowledge on the runtime of the different data structures. This thesis provides empirical measurements on the runtime of the data structures so that the results may be referred to, when making the decision which data structures to use. It is important to keep in mind, that all attempts to determine the performance threshholds are subject to the limitations of the hardware, operating system and other base conditions. Furthermore, the observations and conclusions are about the runtime and as such do not make claims about other attributes that may make a certain data structure more suitable for a specific task than another. The rest of this thesis is structured as follows:

- A summary of the observed data structures is provided in section 4
- The procedure used to collect data about the execution of the insertion/access operation is described in section 5.
- The collected data is represented in illustrations of graphs in section 6.
- To anwer the question about performance, the conclusions are formed in section 7.

2 Motivation

At the moment, information on runtime and memory usage of Prolog data structures is sparse. This thesis is motivated by the desire to fill the knowledge gap on runtime of data structures in Prolog. And in consequence to allow to make better informed decisions about the data structures to use from the point of view of performance. In theory the expected number of operations and with that, an estimated runtime of most of the observed data structures, is known. But in praxis, the implementations of both Prolog and the data structures have an influence on the runtime. Mutable data structures like Mutdict and Mutarray are supposed to be very fast, but is that the case regardless of the size of the data structure and the number of operations and are they worth adapting? The Ordered Set provides sorting of the elements it contains, but how does this affect performance compared to a List and are there cases where sorting improves performance? When do the data

[1] https://sicstus.sics.se/index.html

structures overtake each other in performance, are certain data structures preferable for small, medium or large sizes? Furthermore, it is interesting to look for a data structure with the best runtime.

3 Related Work

Data on the runtime and memory usage of the Prolog data structures considered have not yet been made publicly available. Similar studies may exist on data structures in other programming languages.

Carlsson and Mildner [CM12] describe SICStus Prologs development history, system anatomy and design decisions.

Körner, Leuschel, Barbosa, Costa, Dahl, Hermenegildo, Morales, Wielemaker, Diaz, Abreu and Ciatto [KLB+22] provide a comprehensive summary of the history and current state of Prolog, as well as the future prospects of the language. The insights into the workings of the Prolog programming language that this work provides have helped in the writing of this thesis.

Buchholz [Buc22] offers an evaluation of data structures in Prolog for a selection of data points. The limited number of data points limits the validity of conclusions about the performance of the various data structures. This thesis attempts to extend this work by providing a more comprehensive evaluation.

Shaffer [Sha97] describes techniques for representing data in C++. This includes the implementation and theoretic runtime analysis of (sorted) Lists, AVL Tress, Arrays and Dictionaries.

Barklund and Millroth [BM87] propose a way to integrate complex data structures into Prolog.

Appleby, Carlsson, Haridi and Sahlin [ACHS88] describe the SICStus Prolog garbage collector.

The geometric mean is used in this work to summarize the benchmark results. Fleming and Wallace [FW86] demonstrate why the geometric mean is appropriate for summarizing normalized benchmark results.

4 Background

In this section the observed data structures are described and a short summary of the observed memory parameters is offered.

4.1 Observed Data Structures

The Prolog development system used is SICStus[2] and as such the implementations of the observed data structures come from the corresponding SICStus Prolog libraries. In the following subsections, short summaries as well as performance predictions of each of the observed data structures are presented.

Lists are singly linked Lists, a finite sequence of elements[3]. The observed List is unordered and duplicates are allowed. It is expected that the List performs well for the insertion operation. This is the case because the elements are inserted at the head in constant time without checking for duplicates or following a specific order. In contrast the access operation is expected to perform very bad. To access an element the List needs to be iterated over and the time needed to do that increases very fast with the list size. A linear runtime is expected for the access operation and a constant runtime for the insert operation.

Lists without duplicates are the same as the lists mentioned in the previous section, with the difference that they implement their own insertion method. The list without duplicates is therefore observed only when benchmarking the insertion process. With the requirement to avoid duplicates, an iteration over the list is needed during the insertion operation. The runtime of the insertion operation is heavily impacted by that. For both the access and insertion operation a linear runtime is expected.

Ordered Sets are ordered lists without duplicates[4]. They can be expected to perform well if the ordering of the elements provides an advantage, since it requires additional runtime to keep the Ordered Set ordered. Otherwise the performance should be similar, but worse than that of a list without duplicates. They are expected to perform well if the access operation is performed with keys that are not contained in the data structure. Therefore, it is also observed how the access operation behaves when it is performed on a data structure with an element that is not contained in the data structure. For both the access and insertion operation a linear runtime is expected.

AVL Trees are tree implementation of "association lists"[5]. In theory, AVL Trees perform logarithmic for both the access and insertion operation. It is expected that the logarithmic behavior is represented in the runtime of the access and insertion operation.

[2] https://sicstus.sics.se/

[3] https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/ref_002dsyn_002dcpt_002dlis.html#ref_002dsyn_002dcpt_002dlis

[4] https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/lib_002dordsets.html#lib_002dordsets

[5] https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/lib_002davl.html#lib_002davl

Mutdicts are unordered key-value collections, that use a hash table representation[6]. Mutdicts are one of the observed mutable data structures and therefore it is expected that they perform well for both the insertion and access operation. The proposed runtime of both operations is constant.

Mutarrays are a SICStus Prolog implementation of arrays[7]. They are limited to integer values as keys and as such are not observed for other key types. Because arrays should throw an exception if an index out of bounds is accessed, it is not of interest to observe the access operation on a Mutarray with keys that are not contained in it (as described in paragraph Ordered Sets 4.1). Mutarrays are one of the observed mutable data structures and therefore it is likely that they perform well for both the insertion and access operation. The expected runtime of both operations is constant.

Blackboards are a per-module repository to store elements as key-value pairs[8]. Blackboards are limited to integer and atom values as keys and as such are not observed for other key types. It is expected that they perform well for both the insertion and access operation. The likely runtime of both operations is constant.

Assert allows to add dynamic clauses to the database[9]. The insertion operation is expected to perform well, unlike the access operation, whose performance may vary depending on the key type. This is the case because Assert uses first-argument indexing[10]. A constant runtime is likely for the insert operation. For the access operation with atoms or integer values as keys, a constant runtime is also predicted. When complex terms are used as keys, the runtime may be linear depending on how many similar terms are contained in the data structure.

4.2 Memory Parameter

Memory management, including the garbage collector, can affect runtime, so some memory parameters are also considered. The three parameters observed are *garbage collector calls*, *heap bytes used* and *global stack bytes used*.

[6] https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/lib_002dmutdict.html#lib_002dmutdict

[7] https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/lib_002dmutarray.html#lib_002dmutarray

[8] https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/ref_002dmdb_002dbbd.html#ref_002dmdb_002dbbd

[9] https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/mpg_002dref_002dassert.html#mpg_002dref_002dassert

[10] https://sicstus.sics.se/sicstus/docs/4.6.0/html/sicstus/Indexing-Overview.html

Garbage collector calls indicates how often the garbage collector had to be used. The numbers shown include the number of times the garbage collector is invoked when the data structure is initialized and when the access/insert operation is executed.

Heap bytes used refers to the bytes occupied by symbol tables and the like. Observations about the heap are limited to the Blackboard and Assert data structures, since the other observed data structures do not use the heap. The numbers shown include the difference of the bytes used before and after initializing the data structure and before and after performing the insertion/access operation.

Global stack bytes used is the amount of bytes occupied by compound terms[11]. The numbers shown include the difference of the bytes used before and after initializing the data structure and before and after performing the insertion/access operation.

5 Procedure

This section describes the procedure used to generate the results. This includes a description of the data patterns used, how random elements are generated, and how the benchmark is performed for access and insert operations. Benchmarks are executed in Prolog. Figures are build in Python using the libraries numpy, mathplotlib and pandas. The program code, benchmarks as well as all figures can be viewed in the thesis repository[12]. Benchmarks are stored as csv-files, figures as png-files. To minimize the influence of measurement inaccuracies, each benchmark is repeated five times and averaged using the geometric mean. To account for zero values when using the geometric mean, 1 is added to the measurements before the calculation and 1 is subtracted from the result after the calculation. Negative values could occurr when benchmarking the difference in global stack bytes used. These values were set to zero.

5.1 Random Number Generation

To simulate random accesses on the data structures random numbers are needed. Random numbers are generated using the random library of SICStus-Prolog. To obtain comparable and repeatable results, the same random seed is used during benchmarking. Every time one or more random selections are performed, the random seed used is 42 if no other number is mentioned.

[11] https://sicstus.sics.se/sicstus/docs/3.12.4/html/sicstus/Compound-Terms.html
[12] https://gitlab.cs.uni-duesseldorf.de/stups/abschlussarbeiten/christoph-ludolf-bachelor

Listing 1: Definition of a data structure.

```
1: data_structure(my_datastructure, myds_new, myds_insert,
2:                          myds_access, myds_delete, myds_clear).
3: myds_new(NewDataStructure, N):- ... .
4: myds_insert(L, DataStructureIn, DataStructureOut) :- ... .
5: myds_access(L, DataStructure):- ... .
6: myds_delete(L, DataStructureIn, DataStructureOut):- ... .
7: myds_clear(DataStructure, DataStructureClear):- ... .
```

Listing 2: Definition of a data pattern.

```
1: data_pattern(my_datapattern, dataset_mydp, pattern_mydp,
2:                  misspattern_mydp).
3: dataset_mydp(DataSet, N):- ... .
4: pattern_mydp(HitPattern, DataSet, N):- ... .
5: misspattern_mydp(MissPattern, DataSet, N):- ... .
```

5.2 Data Structures

Data structures are defined by five predicates as shown in listing 1. *myds_new* creates a new empty data structure of size N. The data structure size parameter N is only needed for data structures that are initialized with a constant size like the Mutarray. *myds_insert* performs the insertion operation with all elements from the List L on *DataStructure*. *myds_access* performs the access operation with all elements from the List L on *DataStructure*. *myds_delete* performs the deletion operation with all elements form the List L on *DataStructure*. *myds_clear* removes all entries from *DataStructure*. The main purpose of *myds_clear* is to guarantee that global data structures like Assert are emtpy at the beginning of the next benchmark. In the special case of Blackboard this does not work because at the moment it is not possible to list elements contained in it. To empty the Blackboard after each iteration a special condition is included in the bench loop.

5.3 Data Patterns

In order to build the data structures and perform the various operations, data sets are required. These data sets are created using the data patterns described in this section. Each data pattern is defined by three predicates, as displayed in listing 2. *dataset_mydp* creates *DataSet* of size N with unique keys that are used as initial entries for a data structure. Second, *pattern_mydp* initializes *HitPattern*, a List of keys of length N that are contained in *DataSet*. The third predicate *misspattern_mydp* initializes *MissPattern*, a List of keys of length N that are not contained in *DataSet*. This allows to initialize a data structure with the keys of *DataSet* and, for example, perform access operations on it using the Lists

created by *pattern_ mydp* and *misspattern_ mydp*.

rannumbers is used when integer values are needed as keys and the miss percentage is not observed. With it the *DataSet*, a *HitPattern* and a *MissPattern* is created. The data pattern *rannumbers* for a data structure of size N generates a *DataSet* containing each number from *[1, 2, .., N]* in a random order. To generate *HitPattern* of length M an integer is randomly selected from *[1, 2, ..., N]*. This is repeated M-times. To generate *MissPattern* of length M a number is randomly selected from *{-1, N+1}*. This is repeated M-times.

rannumbers10 is used when integer values are needed as keys and when the miss percentage is observed. With it the *DataSet*, a *HitPattern* and a *MissPattern* is created. The *DataSet* of *rannumbers10* for a data structure of size N, contains all multiple of ten that are included in *[10, 20, ..., N*10]* in a random order. This allows to perform the access operations to access keys, that are not contained in the data structure, but would be placed between keys that are contained in the data structure. To generate *HitPattern* of length M an integer is randomly selected from *[1, N]* and multiplied by *10*. This is repeated M-times. To generate *MissPattern* of length M first, an integer is randomly chosen from *[1, N+1]*. Then the chosen number is multiplied by *10* and finally subtracted by an integer randomly selected from *[1, 2, ..., 9]*. This is repeated M-times.

ranatoms is used when atoms are needed as keys. With it the *DataSet*, a *HitPattern* and a *MissPattern* is created. Each atom is generated using a different integer value as a seed. A randomly generated atom is of the shape:

myatom_*character1 character2 seed*

character1 and *character2* are the respective ASCII characters of two randomly selected numbers using *seed* as the random seed. *DataSet* for a data structure of size N consists of N atoms, so that each uses a different integer from *[1, 2, ..., N]* as a random seed. To generate *HitPattern* of length M, an atom term is generated using an integer randomly selected from *[1, 2, ..., N]* as the random seed. This is repeated M-times. To generate *MissPattern* of length M, an atom term is generated using an integer randomly selected from *[N+1, N*2]* as the random seed. This is repeated M-times.

rancomplex is used when complex terms are needed as keys. With it the *DataSet*, a *HitPattern* and a *MissPattern* is created. Each complex term is generated using a different integer value as a seed. A randomly generated complex term is of the shape:

term_*character1*(term_*character2*(*seed*,*number2*,*number1*))

number1 and *number2* are random numbers selected from *[97, 98, ..., 122]* using *seed* as the random seed. *character1* and *character2* are the respective ASCII characters of the

values *number1* and *number2*. *DataSet* for a data structure of size N consists of N complex terms, so that each uses a different integer from *[1, 2, ..., N]* as a random seed. To generate *HitPattern* of length M, a complex term is generated using an integer randomly selected from *[1, 2, ..., N]* as the random seed. This is repeated M-times. To generate *MissPattern* of length M, a complex term is generated using an integer randomly selected from *[N+1, N*2]* as the random seed. This is repeated M-times.

5.3.1 Values

When needed, Values are always the boolean value *True*. The impact of different value types and sizes on the performance is not observed.

5.4 PC Specs

The hardware used to perform the benchmarks on consists of an Intel i9-9900K CPU and 32 GB DIMM 2400 MHz RAM. The Operating System is Windows 10 x64.

5.5 Prolog Version

SICStus Prolog Version 4.7.1 was used.

5.6 Output

Benchmark results are saved as csv files. The naming of the csv files follows the pattern:

DataStructureName_ _ BenchPredicate_ _ DataPatternName_ _ OptionalInfo.csv

Each row of the output files starts with the sample size, followed by the bench results. Because benches are repeated a number of times, each line may contain multiple results per sample size. The column names are numbered to indicate repetitions.

5.7 Benchmark Access

To benchmark the access operation, first the three lists mentioned in listing 2 named *DataSet*, *HitPattern* and *MissPattern* are initialized. One of the data patterns described in section 5.3 is used as *my_ datapattern*. *DataSet* contains the initial elements of the data structure. Both *HitPattern* and *MissPattern* are used as inputs for *myds_ access*, mentioned in listing 1. *HitPattern* is made up of keys that are contained in *DataSet*. *MissPattern* is made up of keys that are not contained in *DataSet*. After the Lists are created, the

following is performed five times: First the data structure is initialized with *myds_new* and the elements from *DataSet* are inserted using *myds_insert*. Next the access operation is performed using *myds_access* with *HitPattern* and *MissPattern* as inputs. Several benchmarks are performed at each iteration, including but not limited to the time required to access the key sequences. Finally *myds_delete* and *myds_clear* are used to empty the data structure.

5.8 Benchmark Insert

The insert benchmark is performed similiar to the access benchmark. First the two of the three lists mentioned in listing 2 named *DataSet* and *HitPattern* are initialized. One of the data patterns described in section 5.3 is used as *my_datapattern*. *DataSet* contains the initial elements of the data structure. *HitPattern* is used as the input for *myds_insert*, mentioned in listing 1. *HitPattern* is made up of keys that are contained in *DataSet*. After the Lists are created, the following is performed five times: First the data structure is initialized with *myds_new* and the elements from *DataSet* are inserted using *myds_insert*. Next the insertion operation is performed using *myds_insert* with *HitPattern* as input. Several benchmarks are performed at each iteration, including but not limited to the time required to insert the key sequences. Finally *myds_delete* and *myds_clear* are used to empty the data structure.

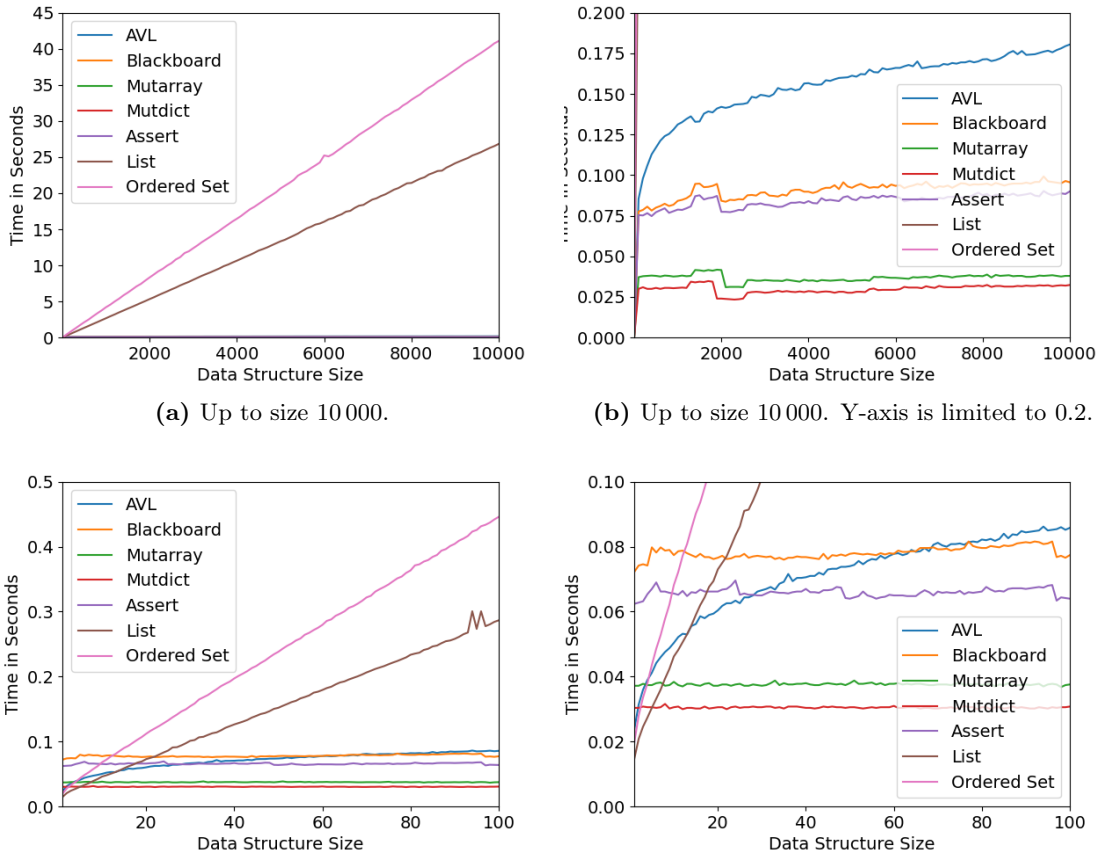
6 Results

In this chapter, the data collected is visualized. Data was gathered on the access and insertion operation. The sub caption of each figure mentions the observed parameters and the used data pattern. See Section 5.3 for a description of the data patterns.

6.1 Access Operation

The observed parameters for the access operation are data structure size, operation count and miss percentage. Data structure size describes the count of elements contained in the data structure. Operation count is the sum of access operations performed in total. The operation count is equal to the sum of the lengths of *HitPattern* and *MissPattern*, mentioned in listing 2. Miss percentage describes, the percentage of the performed access operations that access keys, which are not contained in the data structure. In cases where the miss percentage is greater than 0, the data structure Mutarray is not considered. This is the case because an access of a key not contained in a Mutarray should result in an error. The observed key types are integer values, atoms and complex terms.

6.1.1 Data Structure Size: Integer as Keys



(a) Up to size 10 000.

(b) Up to size 10 000. Y-axis is limited to 0.2.

(c) Up to size 100.

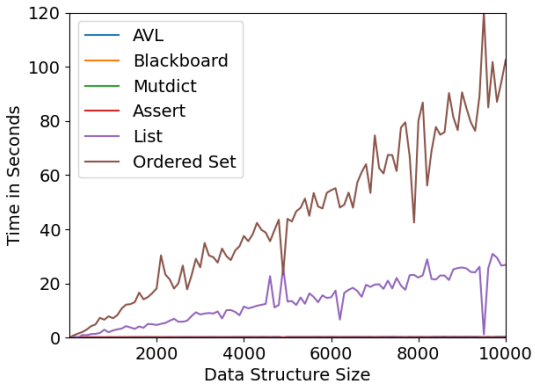
(d) Up to size 100. Y-axis is limited to 0.1.

Figure 1: Access Operation with increasing data structure size paired with constant access operation count (1 000 000). The used data pattern is rannumbers described in section 5.3.

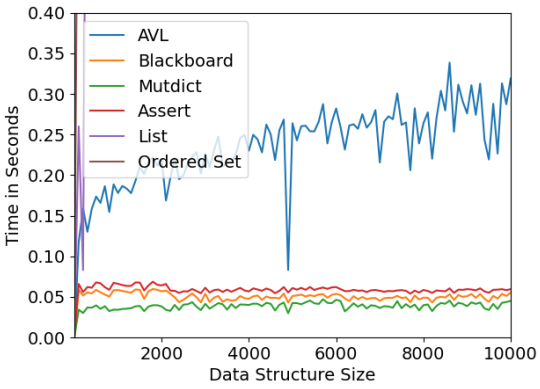
Figure 1 displays the access operation performed with integer values as keys, a constant number of operations and increasing data structure size. The miss percentage is zero. An overall picture of the performance for large data structures is drawn by figure 1a and figure 1b. Figure 1c is a total overview of the graphs from data structure size 1 to 100. Figure 1d is a zoomed in view to highlight the intersection points of the graphs. A closer look reveals that the Ordered Set and List cut through the other data structures early on with a constant increase in runtime. With a runtime of around 40 seconds at data structure size 10 000 the Ordered Set is the slowest data structure displayed. It is followed by the List with a runtime of around 25 seconds at data structure size 10 000. The Ordered Set shows a better runtime than the other data structures until it reaches data structure size 1 to 3. From than on it cuts through the other graphs in a short span and leaves the rest behind

at around data structure size 12. The List displays a better runtime than the other data structures until it reaches data structure size 5. From than on it cuts through the other graphs in a short span and leaves the rest behind at around data structure size 24. The AVL Tree grows logarithmically and thus places itself in the long term between the constant and constant growth data structures. It performs better than most data structures at around data structure size 20 to 25, Mutarray and Mutdict beeing the exception. At data structure size 10 000 the runtime of the AVL Tree is around 0.18 seconds. The runtime of Blackboard and Assert is almost constant. It shows a slight increase, being about 0.085 seconds for a data structure size of 10 000. The runtime of Mutarray and Mutdict is constant and is about 0.035 seconds for data structure size 10 000.

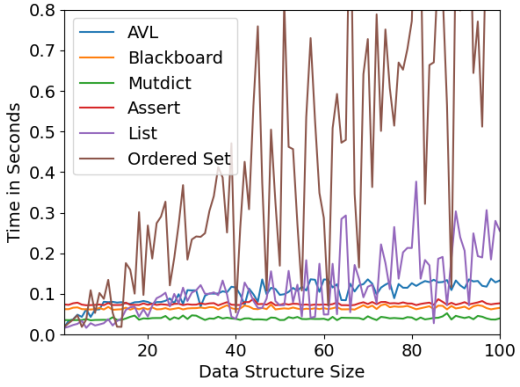
6.1.2 Data Structure Size: Atoms as Keys



(a) Up to size 10 000.



(b) Up to size 10 000. Y-axis limited to 0.4.



(c) Up to size 100.

Figure 2: Access operation with increasing data structure size paired with constant access operation count (1 000 000). The used data pattern is ranatoms described in section 5.3.

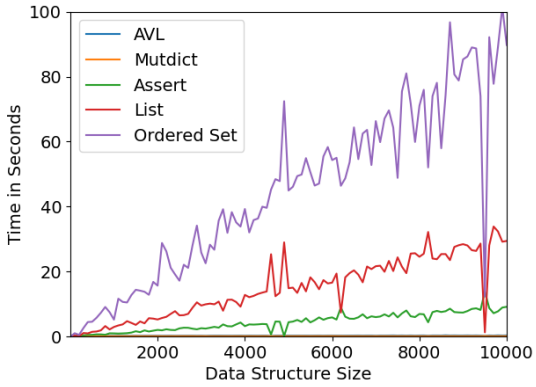
Figure 2 displays the access operation performed with atoms as keys, a constant number of operations and increasing data structure size. The miss percentage is zero. An overall picture of the performance for large data structures is drawn by figure 2a and figure 2b. Figure 2c is a zoomed in view to highlight the intersection points of the graphs. List and Ordered Set both show a constant increase in runtime. With a runtime of around 100 seconds at data structure size 10 000 the Ordered Set is the slowest data structure displayed. It is followed by the List with a runtime of around 25 seconds at data structure size 10 000. The AVL tree displays a logarithmic relation with the data structure size. At data structure size 10 000 the runtime of the AVL Tree is around 0.3 seconds. The runtime of the remaining three data structures, Assert, Blackboard and Mutdict is constant. With a data structure size of 10 000, it is about 0.05 seconds. Ordered Set, AVL Tree and List show heavy fluctuations in runtime. This prevents observations about the intersections.

6.1.3 Data Structure Size: Complex Terms as Keys

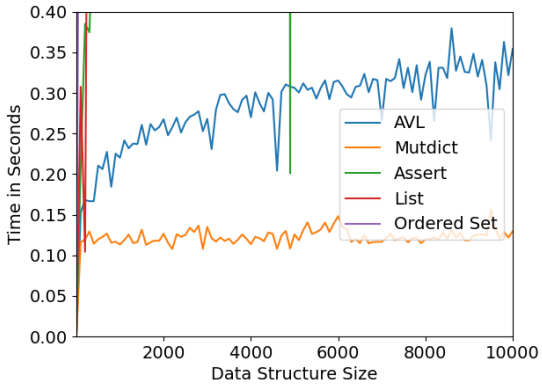
Figure 3 show the access operation performed with complex terms as keys, a constant number of operations and increasing data structure size. The miss percentage is zero. An overall picture of the performance for large data structures is drawn by figure 3a and figure 3b. Figure 3c is a zoomed in view to highlight the intersection points of the graphs. As seen in the figures the runtime of Assert, List and Ordered Set increases constant with data structure size. With a runtime of around 100 seconds at data structure size 10 000 the Ordered Set is the slowest data structure displayed. It is followed by the List with a runtime of around 25 seconds at data structure size 10 000. Next in line is Assert with a runtime of around 10 seconds at data structure size 10 000. AVL shows a logarithmic growth and Mutdict displays a constant runtime. At data structure size 10 000 the runtime of the AVL Tree is around 0.35 seconds. The runtime of the Mutdict is around 0.12 seconds at data structure size 10 000. All data structures show strong fluctuations in runtime. This prevents observations about the intersections.

6.1.4 Miss Percentage: Integer as Keys

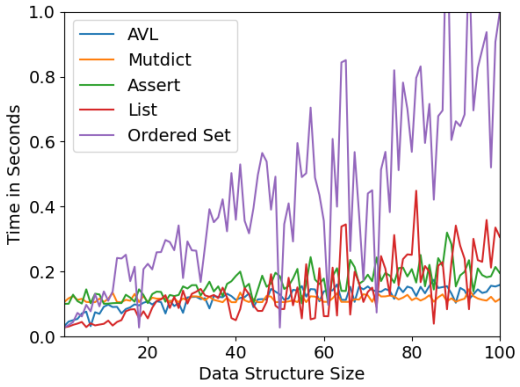
Figure 4 illustrates the access operation performed with integers as keys, a constant number of operations and constant data structure size, but varying miss percentage. Figures 4a and 4b display the varying miss percentage for large data structures. To provide a look at the performance of small data structures, a figure is available that shows the impact of the miss percentage for data structures of size 10 (figure 4c) and 15 (figure 4d). The performance of the Ordered Set increases as the percentage of misses increases. The decrease in runtime is about 10 seconds for data structure size 10 000 at a miss chance of 100 percent. In contrast the performance of the List worsens. The increase in runtime is about 30 seconds for data structure size 10 000 at a miss chance of 100 percent. Ordered Set and List intersect between 45 and 60 percent. This is the case for both small and large data structures. The increasing percentage of misses also influences the performance of the data



(a) Up to size 10 000.



(b) Up to size 10 000. Y-axis limited to 0.4.



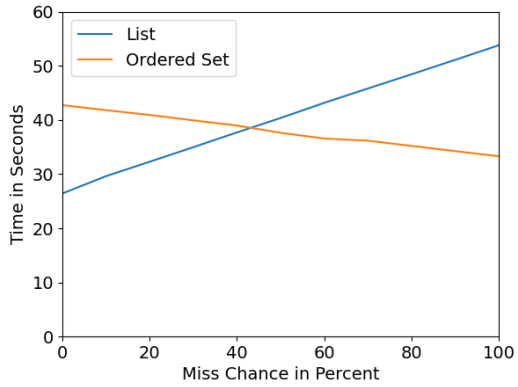
(c) Up to size 100.

Figure 3: Access operation with increasing data structure size paired with constant access operation count (1 000 000). The used data pattern is rancomplex described in section 5.3.

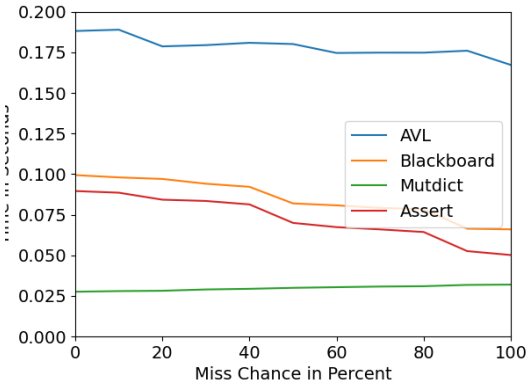
structures Blackboard and Assert for the better. AVL Tree shows a small decrease and Mutdict a very small increase in runtime.

6.1.5 Miss Percentage: Atoms as Keys

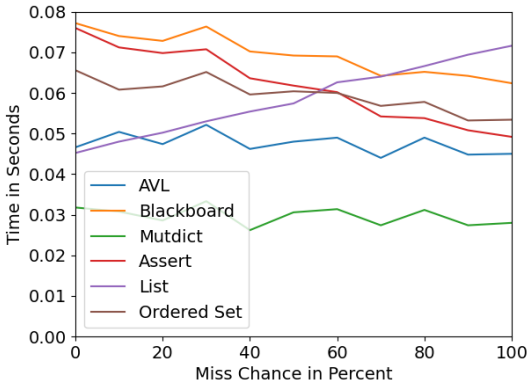
Figure 5 illustrates the access operation performed with atoms as keys, a constant number of operations and data structure size, but varying miss percentage. Figures 5a and 5b display the varying miss percentage for large data structures. To provide a look at the performance of small data structures, a figure is available that shows the impact of the miss percentage for data structures of size 10 (figure 5c) and 15 (figure 5d). The performance of the List decreases as the miss percentage increases. The increase in runtime of the List is about 20 seconds for data structure size 10 000 at a miss chance of 100 percent. When the



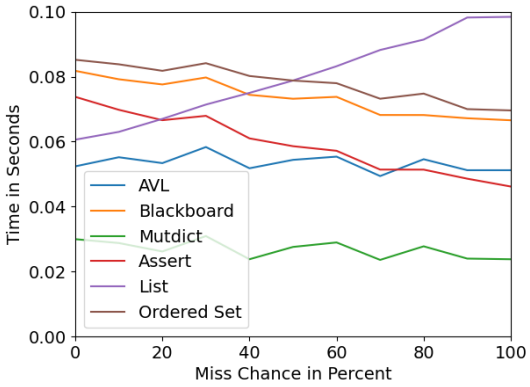
(a) Data structure size 10 000. Ordered Set and List are displayed seperatly.



(b) Data structure size 10 000. The remaining data structures.



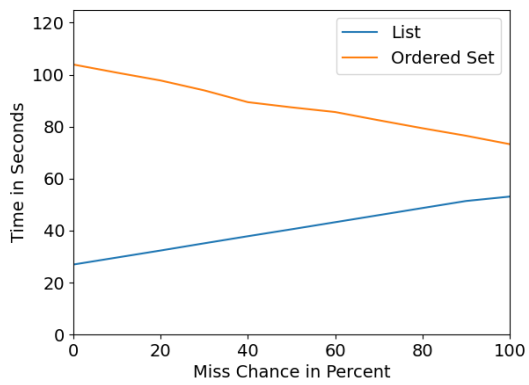
(c) Data structure size 10.



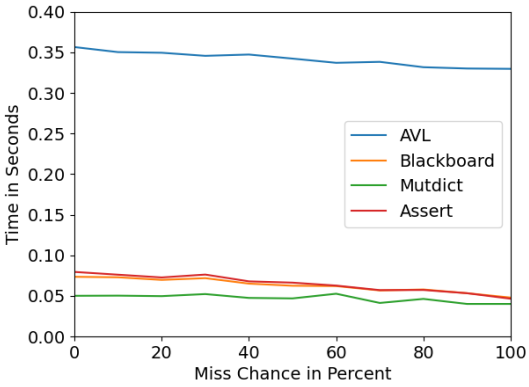
(d) Data structure size 15.

Figure 4: Access operation with increasing miss percentage paired with constant access operation count (1 000 000) and constant data structure size. The used data pattern is rannumbers10 described in section 5.3.

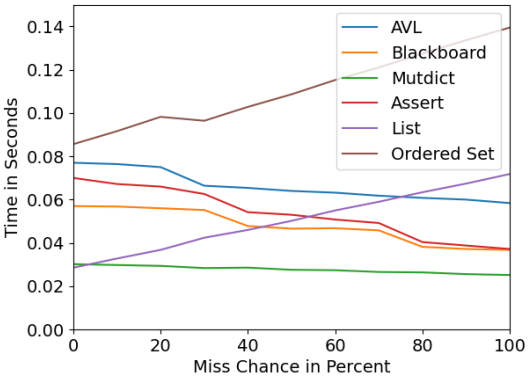
data structure size is 10, the Ordered Set displays a constant increase in runtime. When the data structure size is 15 or the data structure is very large, the runtime of the Ordered Set decreases. For data structure size 15 the Ordered Set is able to outrun the List and AVL Tree for high miss percentage values. For data structure size 10 000, the Ordered Set is no longer able to overtake another data structure in performance. The decrease in runtime of the Ordered Set is about 25 seconds for data structure size 10 000 at a miss chance of 100 percent. The increasing percentage of misses also influences the performance of the data structures Blackboard and Assert for the better. The AVL Tree shows as well as the Mutdict show a very small decrease in runtime.



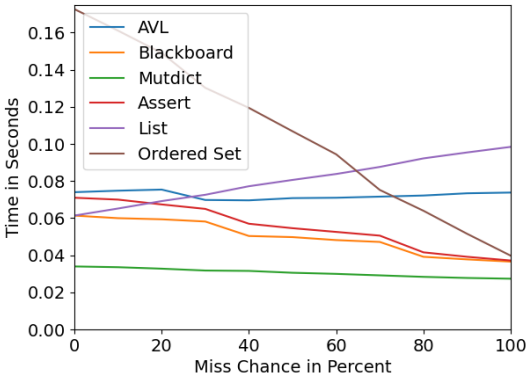
(a) Data structure size 10 000. Ordered Set and List are displayed seperatly.



(b) Data structure size 10 000. The remaining data structures.



(c) Data structure size 10.

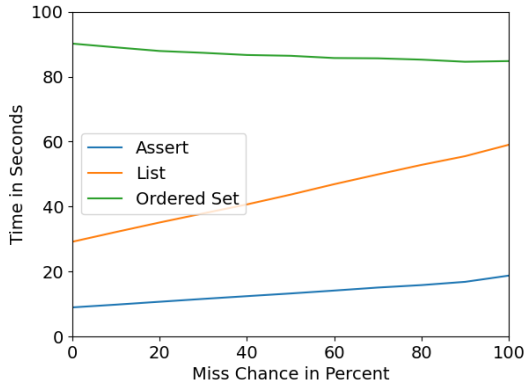


(d) Data structure size 15.

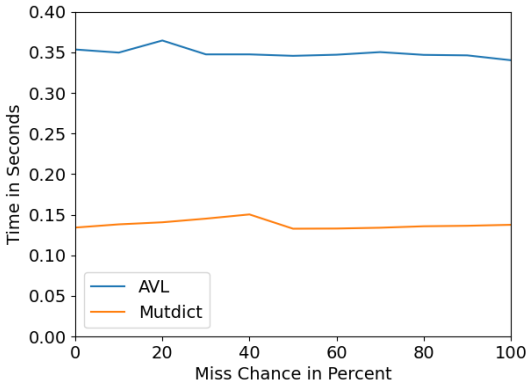
Figure 5: Access operation with increasing miss percentage paired with constant access operation count (1 000 000) and constant data structure size. The used data pattern is ranatoms described in section 5.3.

6.1.6 Miss Percentage: Complex Terms as Keys

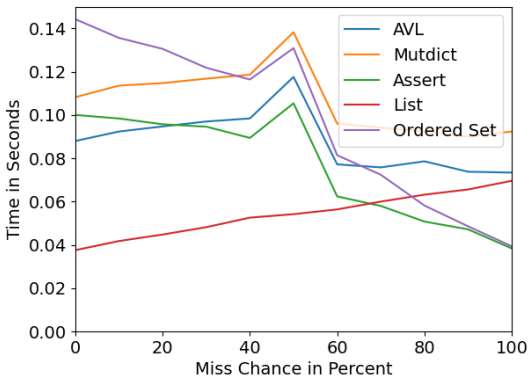
Figure 6 illustrates the access operation performed with complex terms as keys, a constant number of operations and data structure size, but varying miss percentage. Figures 6a and 6b display the varying miss percentage for large data structures. To provide a look at the performance of small data structures, a figure is available that shows the impact of the miss percentage for data structures of size 10 (figure 6c) and 15 (figure 6d). The performance of the Ordered Set increases at higher miss percentages. This increase is high for data structure size 10, but for data structure size 15 and large data structures the change in performance is small. The runtime of the Ordered Set for data structure size 10 000 decreases from around 90 to around 85 seconds. At data structure size 10 the runtime decreases from around 0.14 to around 0.04 seconds. For data structure size 10 and high



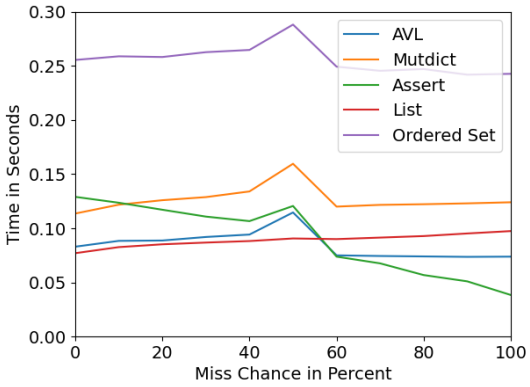
(a) Data structure size 10 000. Ordered Set, List and Assert are displayed seperatly.



(b) Data structure size 10 000. The remaining data structures.



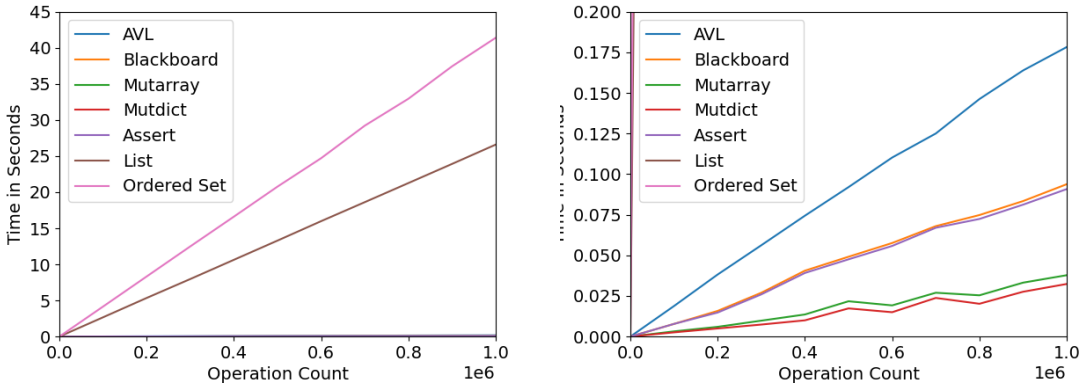
(c) Data structure size 10.



(d) Data structure size 15.

Figure 6: Access operation with increasing miss percentage paired with constant access operation count (1 000 000) and constant data structure size. The used data pattern is rancomplex described in section 5.3.

miss percentage values, the Ordered Set is able to outrun every data structure except Assert in performance. Assert is also able to outrun multiple data structures in performance with increasing miss percentage for small data structure sizes, but shows decreasing performance for large data structure sizes. The AVL Tree manages to overtake the List in performance with increasing miss percentage for data structure size 15. For large data structure sizes no data structure is able to overtake another with increasing miss percentage. The runtime of List increases with the miss percentage. The increase in runtime of the List is about 30 seconds for data structure size 10 000 at a miss chance of 100 percent. The effect of the miss percentage on the AVL Tree and Mutdict is very small.



(a) Total. **(b)** Y-axis limited to 0.2.

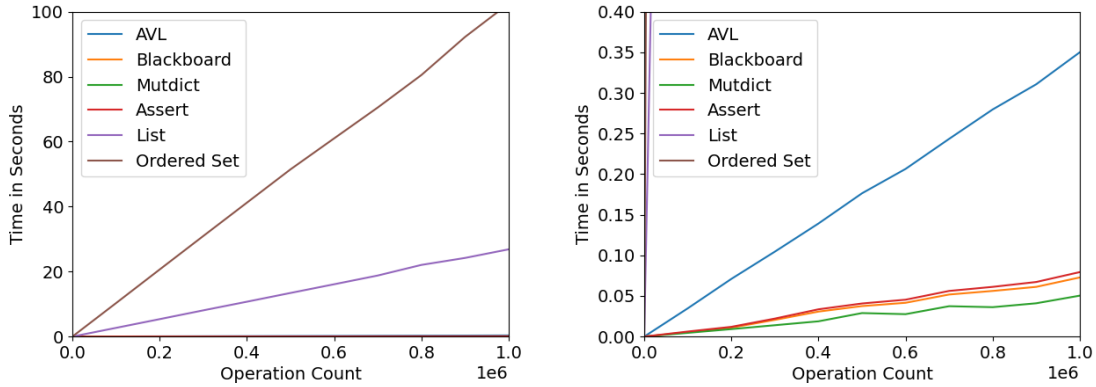
Figure 7: Access operation with increasing operation count paired with constant data structure size (10 000). The used data pattern is rannumbers described in section 5.3.

6.1.7 Operation Count: Integer as Keys

Figure 7 illustrates the access operation performed with integer values as keys, a varying number of operations and constant data structure size. The miss percentage is zero. An overall picture of the performance is drawn by figure 7a. Figure 7b offers a zoomed in view to allow a closer look at the faster data structures. All data structures display a constant increase in runtime. The runtime of Ordered Set and List is very high, reaching over 20 seconds at 1 000 000 operations. Next in line is the AVL Tree with around 0.175 seconds at 1 000 000 operations. With about half that runtime at one million operations, Blackboard and Assert are next. The best runtime is shown by Mutarray and Mutdict, beeing below 0.05 seconds at one million operations.

6.1.8 Operation Count: Atoms as Keys

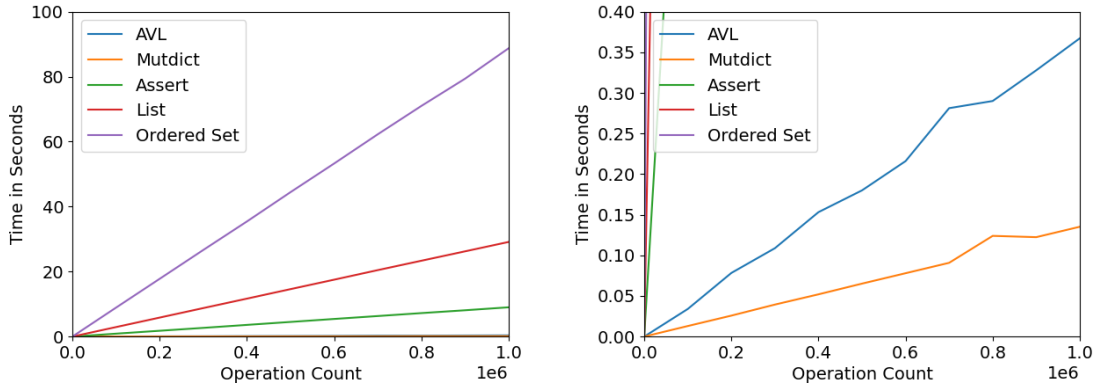
Figure 8 illustrates the access operation performed with atoms as keys, a varying number of operations and constant data structure size. The miss percentage is zero. An overall picture of the performance is drawn by figure 8a. Figure 8b offers a zoomed in view to allow a closer look at the faster data structures. All data structures display a constant increase in runtime. The runtime of Ordered Set and List is very high, with List reaching over 20 and Ordered Set over 90 seconds at 1 000 000 operations. Next in line is the AVL Tree with around 0.35 seconds at 1 000 000 operations. With around 0.06 seconds at 1 000 000 operations Assert and Blackboard are next. The best runtime is shown by Mutdict, beeing below 0.05 seconds at one million operations.



(a) Total. **(b)** Y-axis limited to 0.4.

Figure 8: Access operation with increasing operation count paired with constant data structure size (10 000). The used data pattern is ranatoms described in section 5.3.

6.1.9 Operation Count: Complex Terms as Keys



(a) Total. **(b)** Y-axis limited to 0.4.

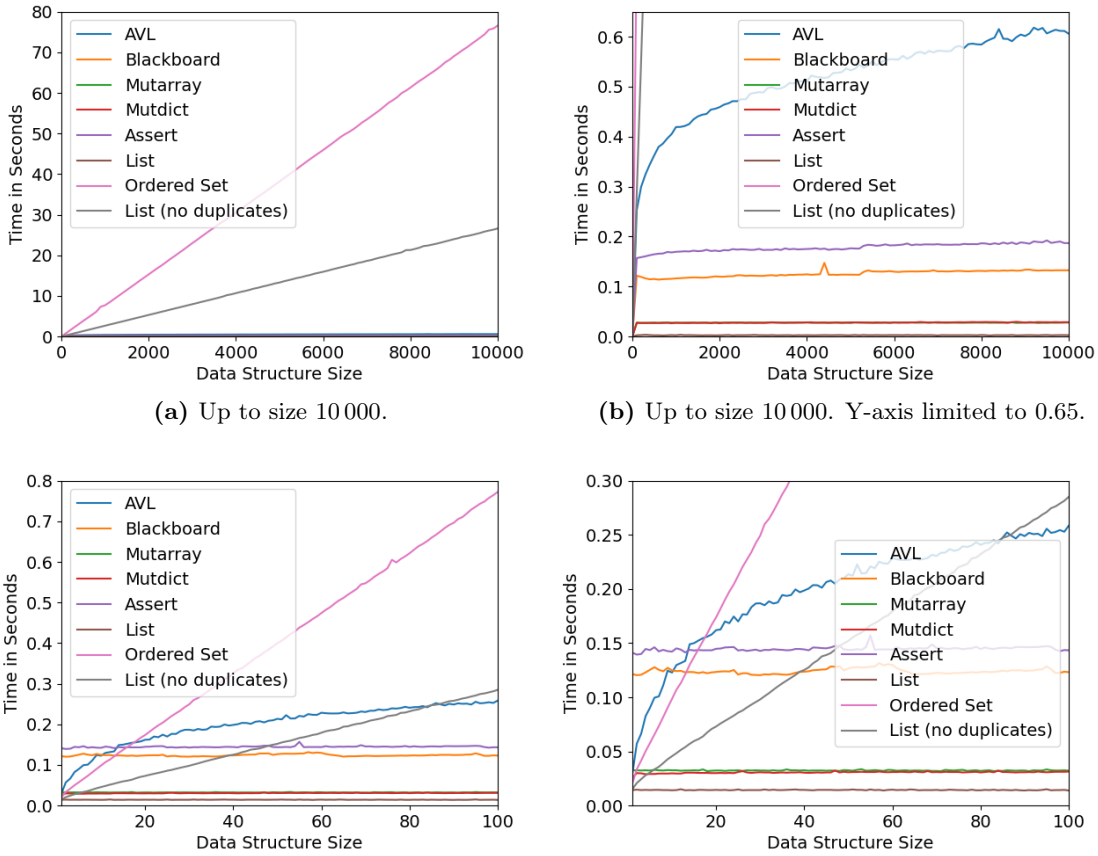
Figure 9: Access operation with increasing operation count paired with constant data structure size (10 000). The used data pattern is rancomplex described in section 5.3.

Figure 9 illustrates the access operation performed with complex terms as keys, a varying number of operations and constant data structure size. An overall picture of the performance is drawn by figure 9a. Figure 9b offers a zoomed in view to allow a closer look at the faster data structures. The miss percentage is zero. All data structures display a constant increase in runtime. The runtime of Ordered Set, List and Assert is very high, with Assert around 9 List around 20 and Ordered Set around 85 seconds at 1 000 000 operations. Next in line is the AVL Tree with around 0.375 seconds at 1 000 000 operations. Mutdict shows the best runtime with less than 0.15 seconds at one million operations.

6.2 Insertion Operation

When performing the insertion operation the observed parameters are data structure size and operation count. Data structure size describes the count of elements contained in the data structure. Operation count is the sum of insertion operations performed in total. The observed key types are integer values, atoms and complex terms.

6.2.1 Data Structure Size: Integer as Keys



(a) Up to size 10 000.

(b) Up to size 10 000. Y-axis limited to 0.65.

(c) Up to size 100.

(d) Up to size 100. Y-axis limited to 0.3.

Figure 10: Insert operation with increasing data structure size paired with constant operation count (1 000 000). The used data pattern is rannumbers described in section 5.3.

Figure 10 displays the insert operation performed with integer values as keys, a constant number of operations and increasing data structure size. An overall picture of the performance for large data structures is drawn by figure 10a and figure 10b. Figure 10c is a total overview of the graphs from data structure size 1 to 100. Figure 10d is a zoomed in view to

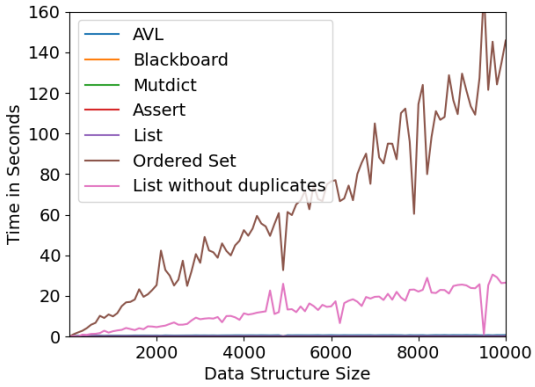
highlight the intersection points of the graphs. Both the Ordered Set and the List without duplicate display a constant increase in runtime. With a runtime of around 75 seconds at data structure size 10 000 the Ordered Set is the slowest data structure displayed. It is followed by List without duplicates at a runtime of around 25 seconds at data structure size 10 000. The Ordered Set performs better than the AVL Tree, Assert and Blackboard until around data structure size 10. The List without duplicates performs better than the AVL Tree, Assert and Blackboard until around data structure size 40. The AVL tree grows logarithmically and thus places itself in the long term between the constant and constant growth data structures. At data structure size 10 000 the runtime of the AVL Tree is around 0.6 seconds. It performs better than Assert and Blackboard until around data structure size 8. At around data structure size 20 the AVL Tree overtakes the Ordered Set in performance and around data structure size 90 the list without duplicates. The remaining data structures Blackboard, Mutarray, Mutdict, Assert and List show a constant runtime. At data structure size 10 000 the runtime of Assert and Blackboard is around 0.125 seconds. With a runtime below 0.05 seconds at data structure size 10 000 Mutarray, Mutdict and List perform the best.

6.2.2 Data Structure Size: Atoms as Keys

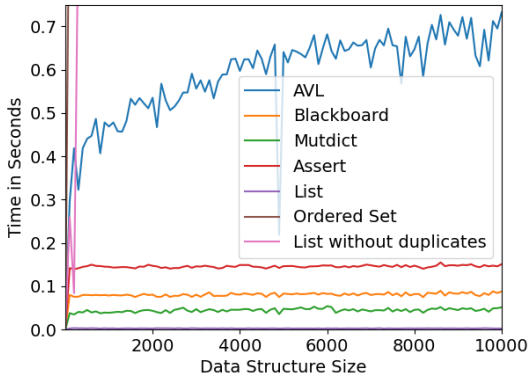
Figure 11 displays the insert operation performed with atoms as keys, a constant number of operations and increasing data structure size. An overall picture of the performance for large data structures is drawn by figure 11a and figure 11b. Figure 11c is a zoomed in view to highlight the intersection points of the graphs. Both the Ordered Set and the List without duplicate display a constant increase in runtime. With a runtime of around 140 seconds at data structure size 10 000 the Ordered Set is the slowest data structure displayed. It is followed by List without duplicates at a runtime of around 25 seconds at data structure size 10 000. The AVL tree grows logarithmically and thus places itself in the long term between the constant and constant growth data structures. At data structure size 10 000 the runtime of the AVL Tree is around 0.7 seconds. The remaining data structures Mutdict, Blackboard, Assert and List show a constant runtime. At data structure size 10 000 the runtime of Assert is around 0.15 seconds. With a runtime below 0.1 seconds at data structure size 10 000 Blackboard, Mutdict and List perform the best. The data structures Ordered Set, AVL Tree and List without duplicates show strong fluctuations. This prevents observations about the intersections.

6.2.3 Data Structure Size: Complex Terms as Keys

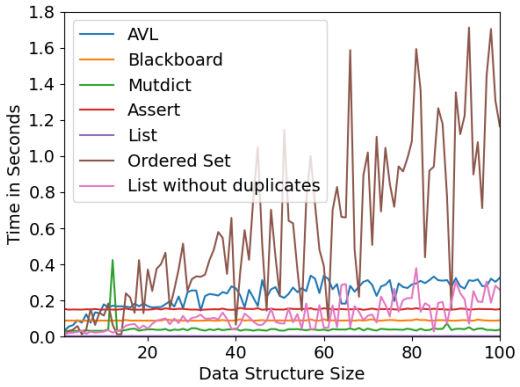
Figure 12 displays the insert operation performed with complex terms as keys, a constant number of operations and increasing data structure size. An overall picture of the performance for large data structures is drawn by figure 12a and figure 12b. Figure 12c is a zoomed in view to highlight the intersection points of the graphs. As seen in the figures the runtime of Ordered Set and List without duplicates increases constant with data structure



(a) Up to size 10 000.



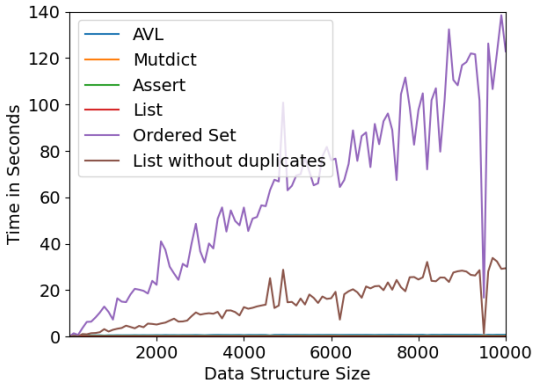
(b) Up to size 10 000. Y-axis limited to 0.65.



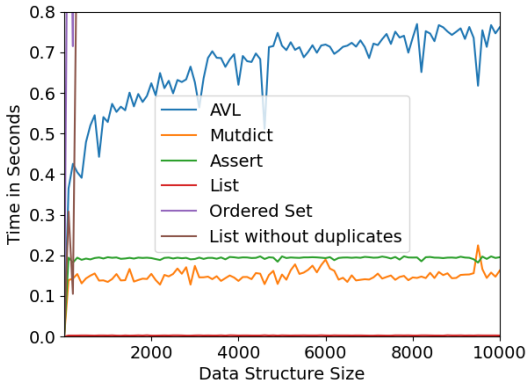
(c) Up to size 100.

Figure 11: Insert operation with increasing data structure size paired with constant operation count (1 000 000). The used data pattern is ranatoms described in section 5.3.

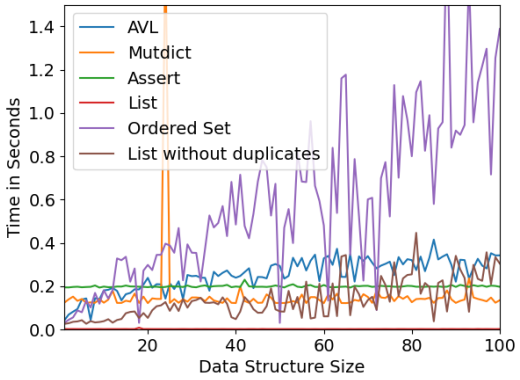
size. With a runtime of around 140 seconds at data structure size 10 000 the Ordered Set is the slowest data structure displayed. It is followed by List without duplicates at a runtime of around 25 seconds at data structure size 10 000. The AVL tree grows logarithmically and thus places itself in the long term between the constant and constant growth data structures. At data structure size 10 000 the runtime of the AVL Tree is around 0.75 seconds. The remaining data structures List, Assert and Mutdict display a constant runtime. At data structure size 10 000 the runtime of Assert and Mutdict is around 0.175 seconds. Mutdict displays a spike in runtime around data structure size 25. With a runtime below 0.02 seconds at data structure size 10 000 List perform the best. The data structures Ordered Set, AVL Tree and List without duplicates show strong fluctuations. This prevents observations about the intersections.



(a) Up to size 10 000.



(b) Up to size 10 000. Y-axis limited to 2.

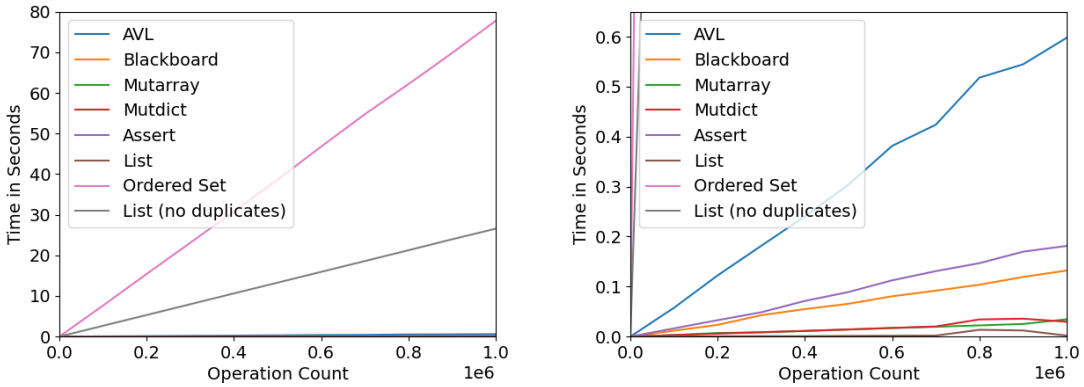


(c) Up to size 100.

Figure 12: Insert operation with increasing data structure size paired with constant access operation count (1 000 000). The used data pattern is rancomplex described in section 5.3.

6.2.4 Operation Count: Integer as Keys

Figure 13 illustrates the insert operation performed with integer values as keys, an increasing number of operations and constant data structure size. An overall picture of the performance is drawn by figure 13a. Figure 13b offers a zoomed in view to allow a closer look at the faster data structures. All data structures show a constant increase in runtime except for the List, which has a nearly constant runtime. The runtime of Ordered Set and List without duplicates is the highest, with the Ordered Set around 75 and List without duplicates around 25 seconds at 1 000 000 operations. Next in line is the AVL Tree with around 0.6 seconds at 1 000 000 operations. With around 0.15 seconds at 1 000 000 operations Assert and Blackboard are next. The runtime of Mutarray, Mutdict and List is less than 0.05 seconds at one million operations.



(a) Total. **(b)** Y-axis limited to 0.65.

Figure 13: Insert operation with increasing operation count paired with constant data structure size (10 000). The used data pattern is rannumbers described in section 5.3.

6.2.5 Operation Count: Atoms as Keys

Figure 14 illustrates the insert operation performed with atoms as keys, an increasing number of operations and constant data structure size. An overall picture of the performance is drawn by figure 14a. Figure 14b offers a zoomed in view to allow a closer look at the faster data structures. All data structures show a constant increase in runtime except for the List, which has a nearly constant runtime. The runtime of Ordered Set and List without duplicates is the highest, with the Ordered Set around 140 and List without duplicates about 20 seconds at 1 000 000 operations. Next in line is the AVL Tree with around 0.7 seconds at 1 000 000 operations. With around 0.1 seconds at 1 000 000 operations Assert and Blackboard are next. The runtime of Mutdict and List is less than 0.05 seconds at one million operations.

6.2.6 Operation Count: Complex Terms as Keys

Figure 15 illustrates the insert operation performed with complex terms as keys, an increasing number of operations and constant data structure size. An overall picture of the performance is drawn by figure 15a. Figure 15b offers a zoomed in view to allow a closer look at the faster data structures. All data structures show a constant increase in runtime except for the List, which has a nearly constant runtime. The runtime of Ordered Set and List without duplicates is the highest, with the Ordered Set around 120 and List without duplicates around 20 seconds at 1 000 000 operations. Next in line is the AVL Tree with around 0.75 seconds at 1 000 000 operations. With around 0.15 seconds at 1 000 000 operations Assert and Mutdict are next. The runtime of List is less than 0.05 seconds at one million operations.

6.3 Memory

In this subsection the observations about the memory usage of the insertion and access operation are presented. The memory parameters considered include global stack bytes used, heap bytes used and garbage collector calls. The global stack and heap bytes used include the difference of the bytes used before and after initializing the data structure and before and after performing the insertion/access operation. The heap is used by Blackboard and Assert, so the heap bytes used parameter is observed only for these data structures. Garbage collector calls include all calls that occur during the initialization of the data structure and during the access/insertion operation.

6.3.1 Global Stack Bytes Used: Access Operation with Integer Values as Keys

Figure 16 displays the global stack usage when performing the access operation, with integer values as keys, varying data structure size and constant operation count. Figures 16a and 16b display the access operation for small and large data structure sizes. All data structures except the List fluctuate strongly in global stack usage. The graphs of Assert and Blackboard overlap and the graphs of Mutdict and Mutarray nearly overlap. All data structures are below 20 000 000 bytes used at any observed data structure size.

6.3.2 Global Stack Bytes Used: Insertion Operation with Integer Values as Keys

Figure 17 displays the global stack usage when performing the insertion operation, with integer values as keys, varying data structure size and constant operation count. Figures 17a and 17b show the insertion operation for small and large data structure sizes. The Ordered Set and AVL Tree fluctuate strongly in global stack usage. The remaining data structures show a slow constant increase in memory usage. Ordered Set and AVL Tree are below 60 000 000 bytes used at any observed data structure size. Blackboard, Assert, Mutarray, Mutdict, List and List without duplicates stay below 20 000 000 bytes used.

6.3.3 Global Stack Bytes Used: Access Operation with Atoms as Keys

Figure 18 displays the global stack usage when performing the access operation, with atoms as keys, varying data structure size and constant operation count. Figures 18a and 18b display the access operation for small and large data structure sizes. All data structures except the List fluctuate strongly in global stack usage. The graphs of Assert and Blackboard overlap. All data structures are below 30 000 000 bytes used at any observed data structure size.

6.3.4 Global Stack Bytes Used: Insertion Operation with Atoms as Keys

Figure 19 displays the global stack usage when performing the insertion operation, with atoms as keys, varying data structure size and constant operation count. Figures 19a and 19b show the insertion operation for small and large data structure sizes. The Ordered Set and AVL Tree fluctuate strongly in global stack usage. The remaining data structures show a slow constant increase in memory usage. Ordered Set and AVL Tree are below 70 000 000 bytes used at any observed data structure size. Blackboard, Assert, Mutdict, List and List without duplicates stay below 20 000 000 bytes used.

6.3.5 Global Stack Bytes Used: Access Operation with Complex Terms as Keys

Figure 20 displays the global stack usage when performing the access operation, with complex terms as keys, varying data structure size and constant operation count. Figures 20a and 20b display the access operation for small and large data structure sizes. All data structures except the List fluctuate strongly in global stack usage. Each data structures is below 18 000 000 bytes used at any observed data structure size.

6.3.6 Global Stack Bytes Used: Insertion Operation with Complex Terms as Keys

Figure 21 displays the global stack usage when performing the insertion operation, with complex terms as keys, varying data structure size and constant operation count. Figures 21a and 21b show the insertion operation for small and large data structure sizes. The Ordered Set and AVL Tree fluctuate strongly in global stack usage. The remaining data structures show a slow constant increase in memory usage. Ordered Set and AVL Tree are below 150 000 000 bytes used at any observed data structure size. Blackboard, Assert, Mutdict, List and List without duplicates stay below 20 000 000 bytes used.

6.3.7 Heap Bytes Used

Figure 22 displays the heap usage for the insertion and access operation. The data patterns used are *rannumbers* 5.3 for integer values, *ranatoms* 5.3 for atoms and *rancomplex* 5.3 for complex terms as keys. Figure 22b, displays the heap bytes used by Assert and Blackboard when performing the insertion operation. Figure 22c allows a closer look at the Blackboard. The heap usage of Assert for atoms and integer values as keys is similar. This also applies to Blackboard. At data structure size 100 Assert uses around 190 000 000 bytes for integer values and atoms as keys and around 240 000 000 bytes when observing complex terms as keys. At data structure size 100 Blackboard uses around 17 500 bytes. Figure 22a, displays the heap bytes used by Assert and Blackboard when performing the access operation.

For atoms and integer values as keys the amount of space used by Assert is similar with around 18 000 bytes at data structure size 100. At data structure size 100 Assert uses around 25 000 bytes when complex terms are used as keys. Blackboard uses around 16 000 bytes at data structure size 100 for both integer values and atoms as keys. All three figures display a constant increase in byte usage for Assert and Blackboard. For both operations displayed by figure 22 Assert uses more bytes than Blackboard.

6.3.8 Garbage Collector Calls: Access Operation with Integer Values as Keys

Figure 23 displays the access operation performed with integer values as keys, constant operation count and varying data structure size. Figure 23a highlights the garbage collector call count for small data structure sizes. Figures 23b and 23c offer a look at the garbage collector calls of larger data structures. The Ordered Set displays a constant increase in garbage collector calls with spikes at around data structure size 1000 and 6000. The maximum amount of garbage collector calls for the Ordered Set is around 12 000 at data structure size 6000. The AVL Tree shows a logarithmic relation between garbage collector calls and data structure size until around data structure size 100. Then the increase in garbage collector calls is linear with around 70 calls at data structure size 10 000. AVL Tree, Blackboard, Mutarray, Mutdict and Assert show a spike in garbage collector calls around data structure size 1000. The graphs of Assert and Blackboard overlap and the graphs of Mutdict and Mutarray are similar. Assert and Blackboard show a linear increase in garbage collector calls from around data structure size 1000 to 7000. After that both Assert and Blackboard display a constant increase in garbage collector calls with around 30 calls at data structure size 10 000. Mutdict and Mutarray display a constant increase in garbage collector calls after the spike at around data structure size 1000. At data structure size 10 000 the amount of garbage collector calls of Mutdict and Mutarray is around 8.

6.3.9 Garbage Collector Calls: Insertion Operation with Integer Values as Keys

Figure 24 displays the insertion operation performed with integer values as keys, constant operation count and varying data structure size. Figure 24a highlights the garbage collector call count for small data structure sizes. Figures 24b and 24c offer a look at the garbage collector calls of larger data structures. The Ordered Set displays a constant increase in garbage collector calls with around 3500 calls at data structure size 10 000. The AVL Tree shows a logarithmic relation between data structure size and garbage collector calls with around 11 garbage collector calls at data structure size 10 000. The remaining data structures stay constant at, or below 1 garbage collector call.

6.3.10 Garbage Collector Calls: Access Operation with Atoms as Keys

Figure 25 displays the access operation performed with atoms as keys, constant operation count and varying data structure size. Figure 25a highlights the garbage collector call count for small data structure sizes. Figures 25b and 25c offer a look at the garbage collector calls of larger data structures. The Ordered Set displays an increase in garbage collector calls with around 2000 calls at data structure size 10 000. The Ordered Set shows strong fluctuations in the amount of garbage collector calls. The graphs of Blackboard and Assert overlap. The garbage collector calls of AVL Tree, Blackboard, Mutdict and Assert spike at around data structure size 3000. After that the number of calls of AVL Tree, Blackboard, Mutdict and Assert show a small constant increase. At data structure size 10 000 the amount of calls of the five data structures is below 10. List stays constant at 0 garbage collector calls.

6.3.11 Garbage Collector Calls: Insert Operation with Atoms as Keys

Figure 26 displays the insertion operation performed with atoms as keys, constant operation count and varying data structure size. Figure 26a highlights the garbage collector call count for small data structure sizes. Figures 26b and 26c offer a look at the garbage collector calls of larger data structures. The Ordered Set displays an increase in garbage collector calls with around 2000 calls at data structure size 10 000. The Ordered Set shows strong fluctuations in the amount of garbage collector calls. The AVL Tree display a logarithmic relation between the number of garbage collector calls and the data structure size. The number of garbage collector calls of the remaining data structures is constant. The amount of calls is below 2 for every observed data structure size. The only exception is the Mutdict with a spike around data structure size 12.

6.3.12 Garbage Collector Calls: Access Operation with Complex Terms as Keys

Figure 27 displays the access operation performed with complex terms as keys, constant operation count and varying data structure size. Figure 27a highlights the garbage collector call count for small data structure sizes. Figures 27b and 27c offer a look at the garbage collector calls of larger data structures. The Ordered Set displays an increase in gabage collector calls, with a spike of 6000 calls around data structure size 9000. At data structure size 10 000 the number of garbage collector calls of the Ordered Set is around 2000. The Ordered Set shows strong fluctuations in the amount of garbage collector calls. The AVL Tree shows a logarithmic relation between data structure size and garbage collector calls, with around 8 calls at data structure size 10 000. Assert displays a slow constant increase in calls. At data structure size 10 000 the number of garbage collector calls of Assert is around 10. Mutdict starts with 0 calls until around data structure size 1000. For each of

the remaining observed data structure sizes, the amount of garbage collector calls is 1. List displays a constant 0 garbage collector calls.

6.3.13 Garbage Collector Calls: Insert Operation with Complex Terms as Keys

Figure 28 displays the insertion operation performed with complex terms as keys, constant operation count and varying data structure size. Figure 28a highlights the garbage collector call count for small data structure sizes. Figures 28b and 28c offer a look at the garbage collector calls of larger data structures. The Ordered Set displays an increase in gabage collector calls, with around 12 000 calls around data structure size 10 000. The Ordered Set shows strong fluctuations in the amount of garbage collector calls. The AVL Tree shows a logarithmic relation between data structure size and garbage collector calls, with around 10 calls at data structure size 10 000. List, List without duplicates and Mutdict display a constant amount of 0 garbage collector calls. The only exception is a spike in the calls to the garbage collector for the Mutdict, which occurs around data structure size 23.

6.4 Other

During the benchmarking of the performance of the data structures, some other results were also collected. These are mentioned in the following subsections.

6.4.1 Bulk Access

During benchmarking, it was tested whether the runtime is affected by the count of access/insert calls done at each step through the lists. Figure 30a compares three implementations of the access operation calls when accessing elements of an Ordered Set. The difference in implementation lies in the number of access operation calls at each recursion step through the list that contains the keys to access. The numbers 1, 10 and 100 in the graph names refer to the number of access operation calls. For example *Ordered Set 10* takes the next ten elements of the list, performs the access operation with each element and then continues through the list. As shown, the runtime of all three implementations is almost the same.

6.4.2 Loop Overhead

During benchmarking, it was tested whether the runtime of the loop containing the access/insert calls affected the collected time. Figure 30b compares the time taken to iterate through the lists that contain the keys to access with the time taken to perform the access operation with the keys contained in the lists. The data structure used is an Ordered Set. *Loop Time* displays the time to iterate through the access lists. *Access Time* displays the

time to iterate over as well as access the keys contained in the access lists. The combined size of the access lists is equal to the operation count. As shown in the figure, the time to iterate through the access lists is much less than the time it takes to access the elements.

6.4.3 AVL with fetch check

During benchmarking, it was tested how a fetch check impacts the runtime. The figures 30c and 30d compare the difference in runtime of the implenentation of the insert call on an AVL Tree with and without fetch check. To perform a fetch check, it is checked if a key-value-pair is already contained by the data structure before inserting it. The key-value-pair is only inserted if it is not already in the data structure. The AVL with fetch check performs better than the AVL without it.

6.4.4 Different Random Number

To test the influence of the selected random number, some benchmarks were performed again with the use of another random number. Figure 31 displays the access operation performed with atoms as keys and constant operation count 1 000 000. Figures 31a, 31b and 31c show the influence of the random numbers 42 and 1234 for varying data structure sizes. As displayed by the figures, the data structures behave similar for the chosen random numbers. Figure 31d is the benchmark displayed in figure 5c repeated with random number 1234. The benchmark was performed to test if the behavior of the Ordered Set for data structure size 10 is a consequence of the chosen random number 42. The behavior is also observed for random number 1234.

6.4.5 Garbage Collection Time of Ordered Set

The figure is included because the Ordered Set invokes the garbage collector a lot. Figure 29a shows the percentage of garbage collection in the runtime of operations performed on the Ordered Set. The data patterns used are *rannumbers* 5.3 for integer values, *ranatoms* 5.3 for atoms and *rancomplex* 5.3 for complex terms as keys. When inserting atoms and complex terms, the garbage collector accounts for about 10% of the runtime. For integer values as keys it is about 20%. When accessing atoms and complex terms, the garbage collector accounts for about 20% of the runtime. For integer values as keys it is about 37%.

7 Conclusions

In this section the conclusions based on the results presented in section 6 are formulated. The conclusions are based on the observed key types and data patterns. Conclusions about

the global stack bytes used are difficult due to the large fluctuations in the bytes used and the possibility of negative values.

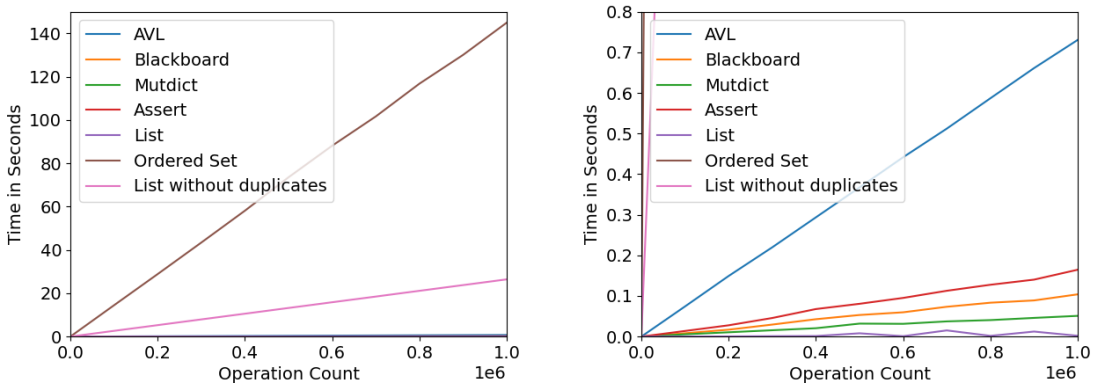
The advantage of the sorting of the Ordered Set comes with a high performance cost, so it performs worst overall for medium to large data structures. Depending on the operation and key type, up to 37% of the runtime is required for the execution of the garbage collector when an Ordered Set is used. Since the runtime of the Ordered Set is very long, this is a considerable amount. The garbage collector could be one factor to increase the performance of the Ordered Set. The performance gain in the access operation offered by sorting is significant only for medium to high miss chance values and very small data structure sizes (10 or lower). An exception is the use of integer values as keys, paired with a high miss chance, where the Ordered Set can outperform a List for larger data structures. In conclusion, the Ordered Set is only preferable for very small data structure sizes and medium to high miss chance values, and it is difficult to outperform the List. For the List, small data structure sizes and a small miss chance is preferred when the runtime is important. A List without duplicates behaves exactly like the List, except that it performs worse for the insertion operation. The runtime of the insertion operation changes from a constant time cost to a constant increase in runtime. The list is slow, but generally performs better than an Ordered Set. The only exception is the access operation with integer values as keys in combination with a high miss percentage. An AVL Tree performs much better than an Ordered Set and a List, but not as good as Mutdict, Mutarray, Blackboard and Assert. The AVL Tree can perform better than Assert, if Assert uses complex terms as keys. The logarithmic behavior of the AVL Tree is confirmed by the benchmarks. Assert is worse at the access operation when complex terms are used. Atoms and integer values as keys are preferable when the runtime of Assert is important. The effects of data structure size on the runtime of Mutdict, Mutarray, and Blackboard are negligible. The question of whether it is worth adapting mutable data structures can be answered with a resounding yes. Mutdict and Mutarray are only topped in performance by a List that allows duplicates when observing the insertion operation. Otherwise, mutable data structures are best in terms of runtime and memory usage. The best performing data structures are Mutdict and Mutarray.

For future analysis, it might be interesting to observe the effects of the *gc_margin* parameter[13], which affects the runtime of garbage collection and global stack allocation. It might be possible to determine an optimal value of *gc_margin* for each data structure, key type and data structure size. Further research could be done on how enabling or disabling the garbage collector effects runtime and memory usage. The benchmarks could be repeated for larger data structure sizes. It might be interesting to observe different data patterns that are not randomized, such as increasing or decreasing integer values.

[13] https://sicstus.sics.se/sicstus/docs/4.5.1/html/sicstus/ref_002dmgc_002dgch.html

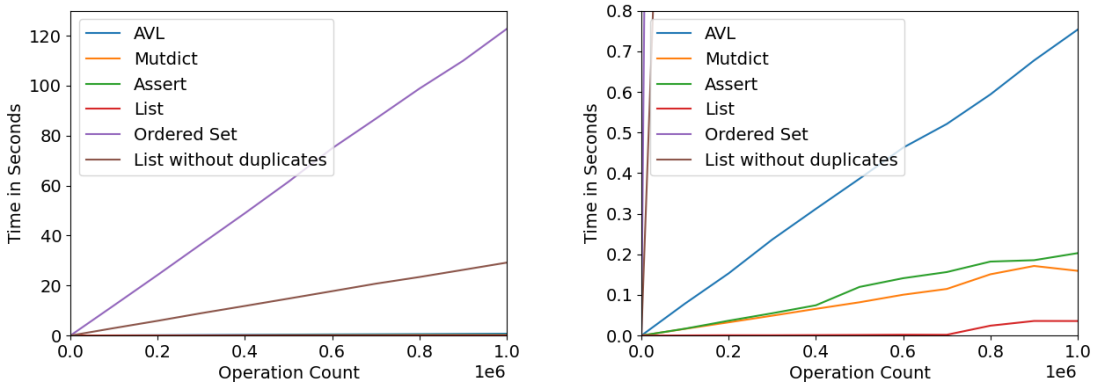
Appendices

A Graphs



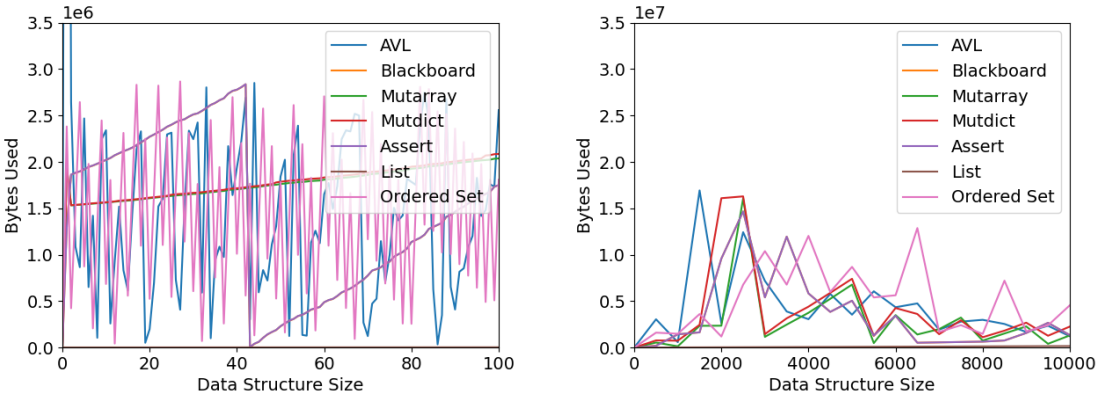
(a) Total. **(b)** Y-axis limited to 0.4.

Figure 14: Insert operation with increasing operation count paired with constant data structure size (10 000). The used data pattern is ranatoms described in section 5.3.



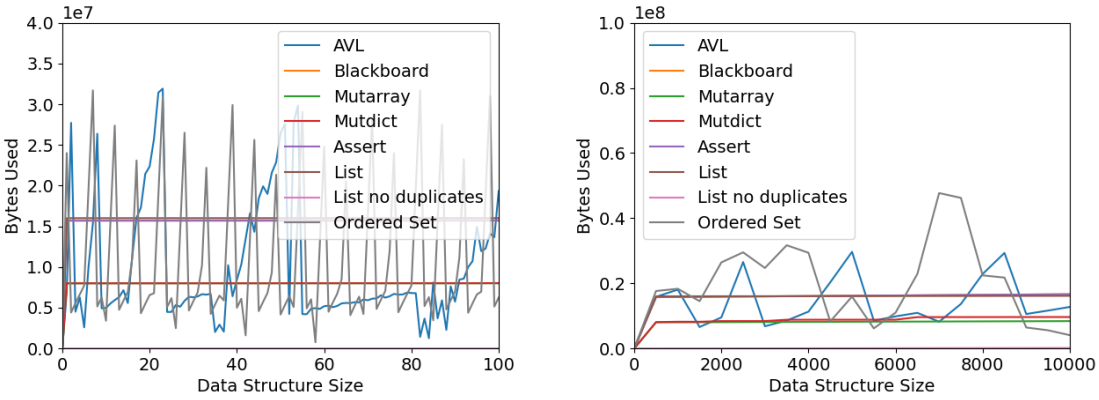
(a) Total. **(b)** Y-axis limited to 0.4.

Figure 15: Insert operation with increasing operation count paired with constant data structure size (10 000). The used data pattern is rancomplex described in section 5.3.



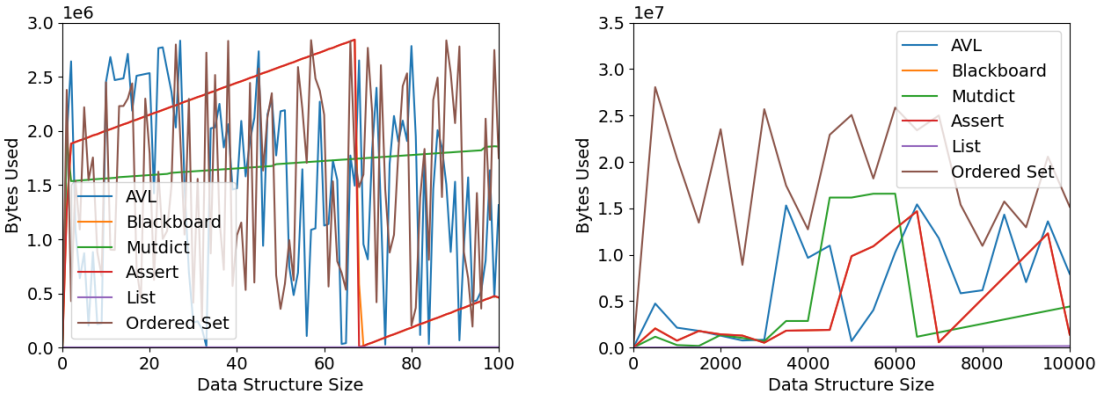
(a) Up to size 100. **(b)** Up to size 10 000.

Figure 16: Global stack bytes used. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rannumbers described in section 5.3.



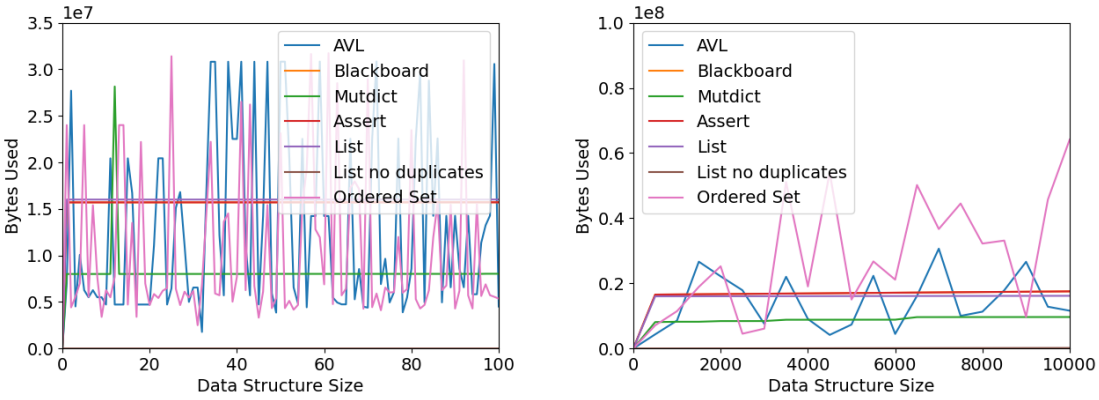
(a) Up to size 100. **(b)** Up to size 10 000.

Figure 17: Global stack bytes used. Insertion operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rannumbers described in section 5.3.



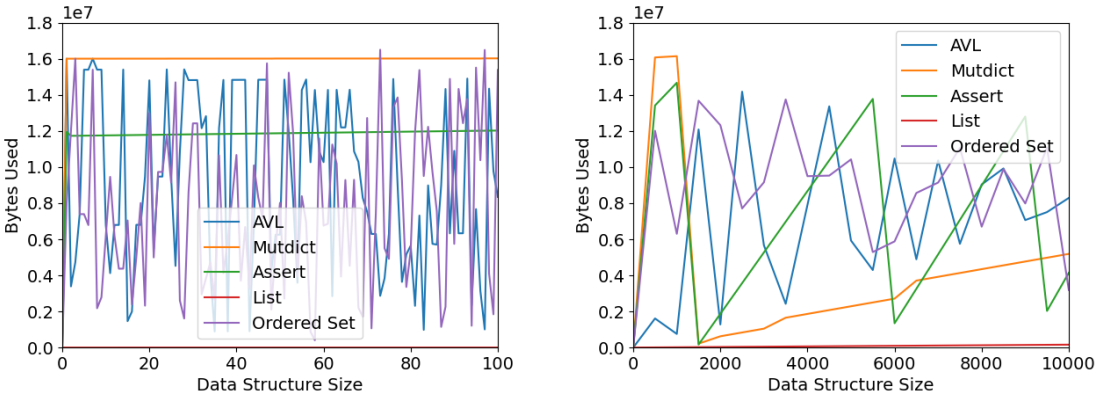
(a) Up to size 100. **(b)** Up to size 10 000.

Figure 18: Global stack bytes used. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is ranatoms described in section 5.3.



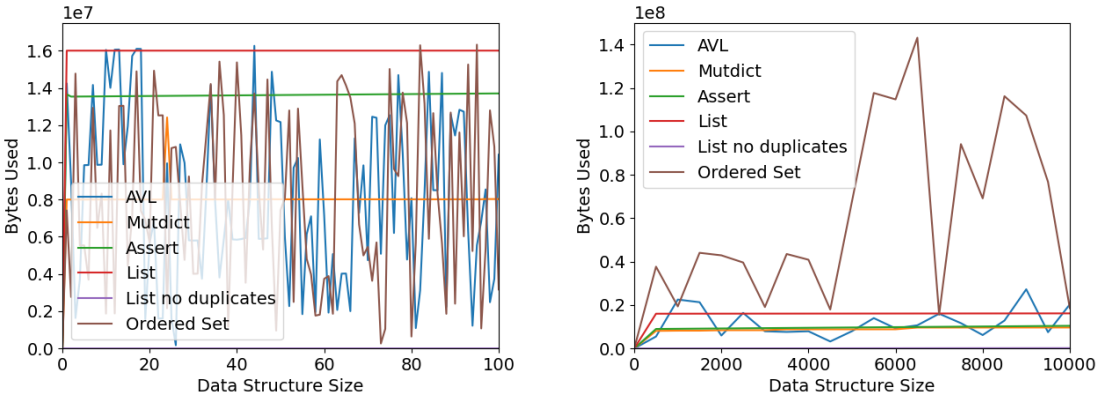
(a) Up to size 100. **(b)** Up to size 10 000.

Figure 19: Global stack bytes used. Insertion operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is ranatoms described in section 5.3.



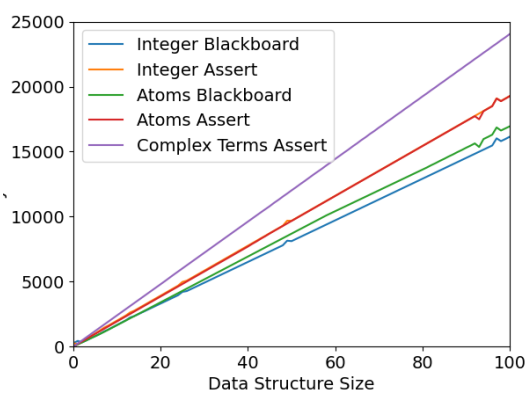
(a) Up to size 100. **(b)** Up to size 10 000.

Figure 20: Global stack bytes used. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rancomplex described in section 5.3.

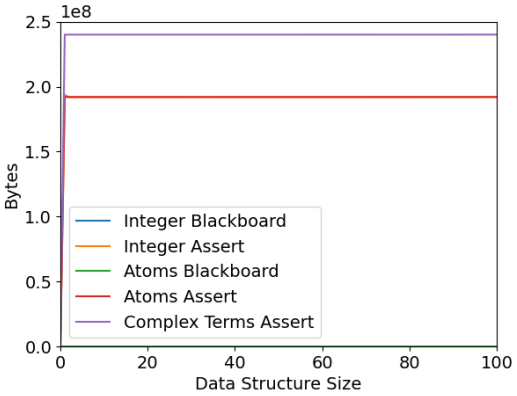


(a) Up to size 100. **(b)** Up to size 10 000.

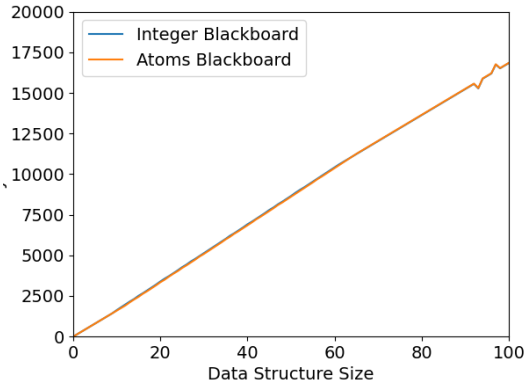
Figure 21: Global stack bytes used. Insertion operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rancomplex described in section 5.3.



(a) Access operation.

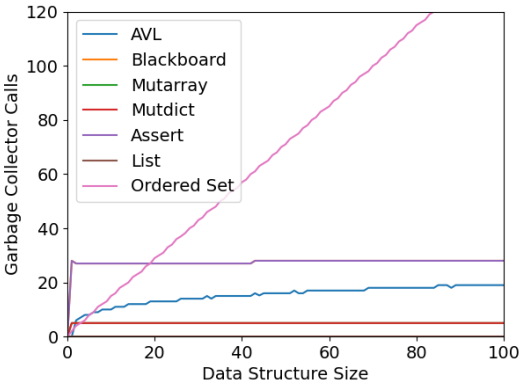


(b) Insertion operation.

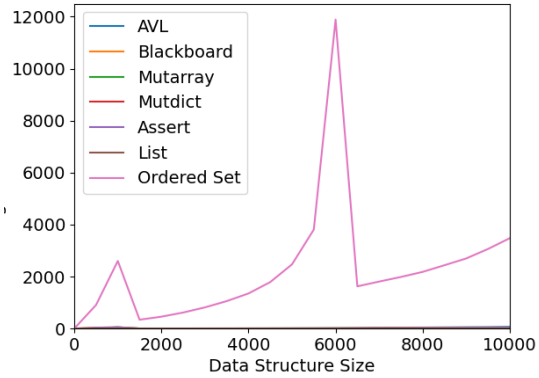


(c) Insertion operation. Y-axis limited to 20 000.

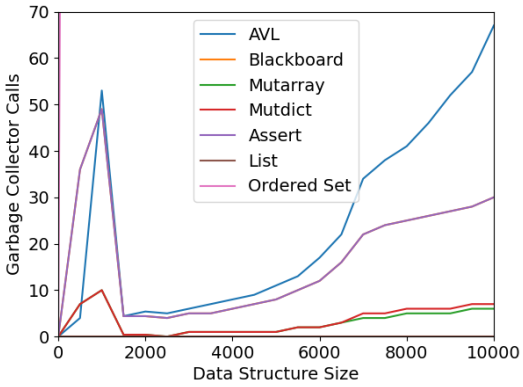
Figure 22: Heap bytes used by Assert and Blackboard. Constant operation count(1 000 000) paired with increasing data structure size.



(a) Up to size 100.

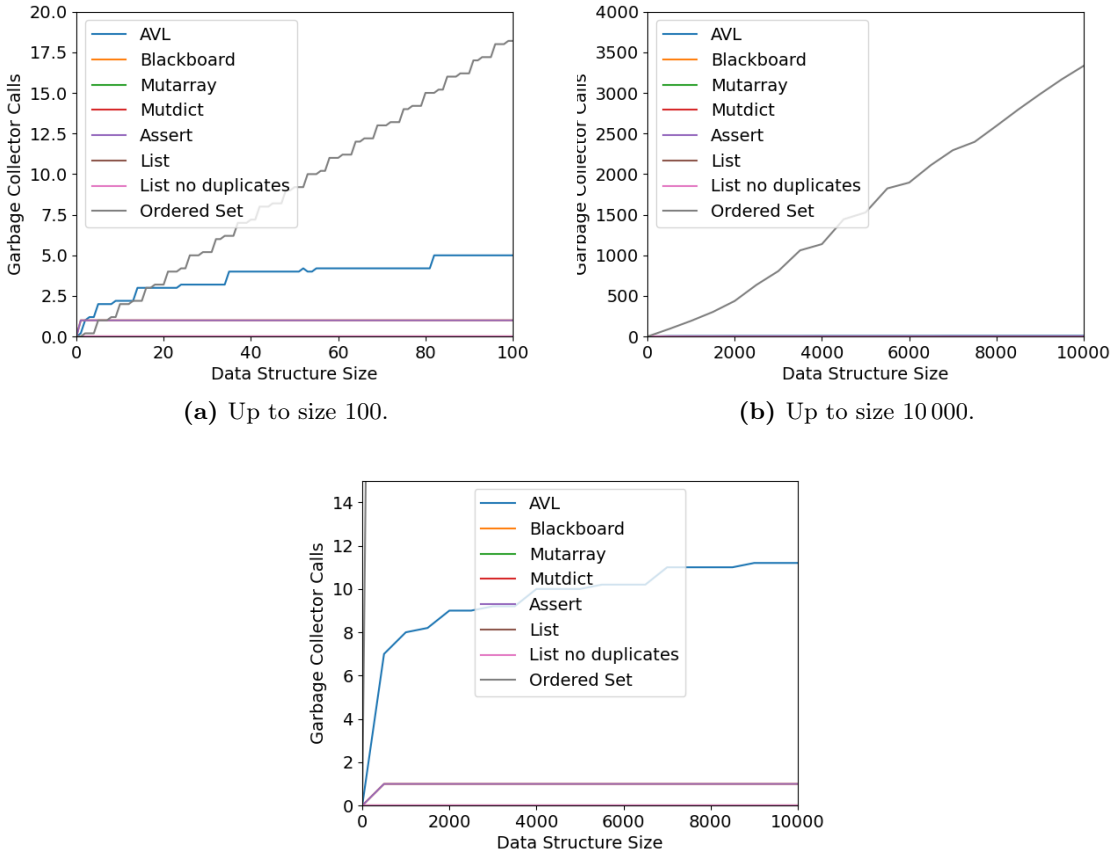


(b) Up to size 10 000.



(c) Up to size 10 000. Y-Axis limited to 15

Figure 23: Garbage collector calls. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rannumbers described in section 5.3.

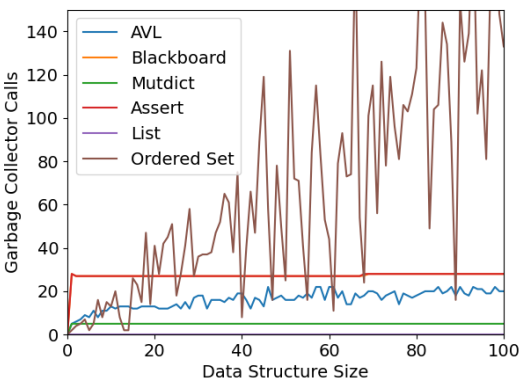


(a) Up to size 100.

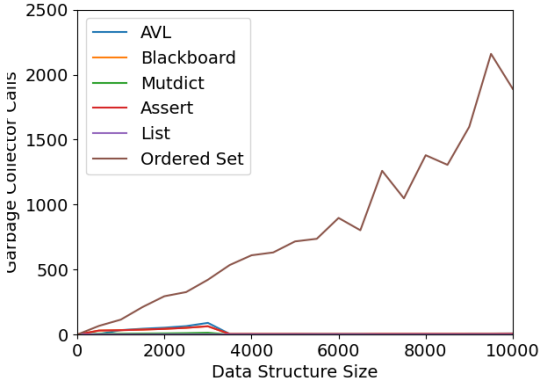
(b) Up to size 10 000.

(c) Up to size 10 000. Y-Axis limited to 15

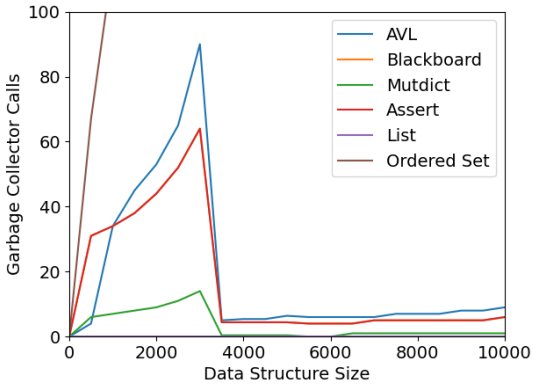
Figure 24: Garbage collector calls. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rannumbers described in section 5.3.



(a) Up to size 100.

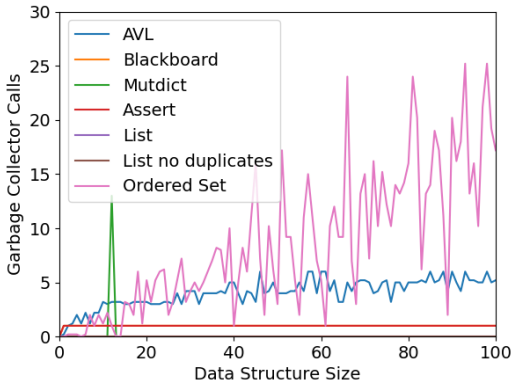


(b) Up to size 10 000.

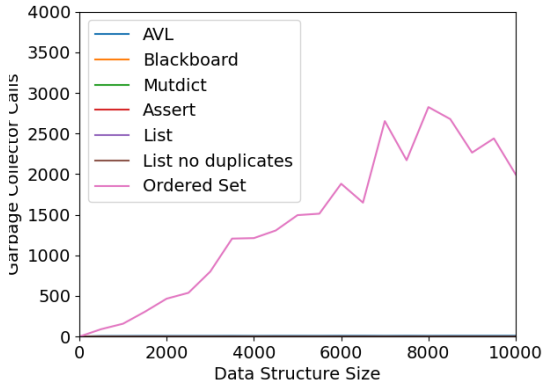


(c) Up to size 10 000. Y-Axis limited to 15

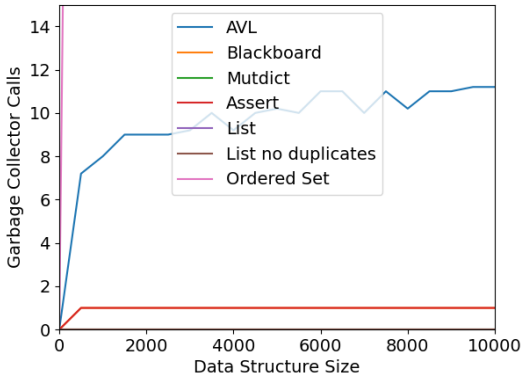
Figure 25: Garbage collector calls. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is ranatoms described in section 5.3.



(a) Up to size 100.

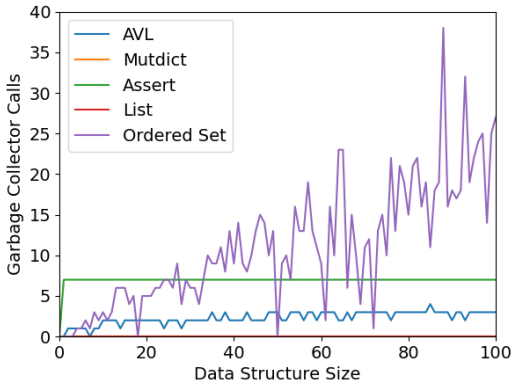


(b) Up to size 10 000.

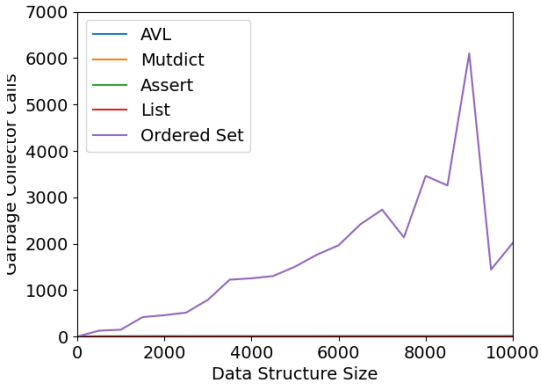


(c) Up to size 10 000. Y-Axis limited to 15

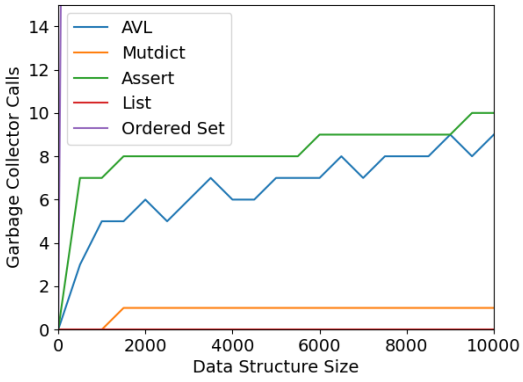
Figure 26: Garbage collector calls. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is ranatoms described in section 5.3.



(a) Up to size 100.

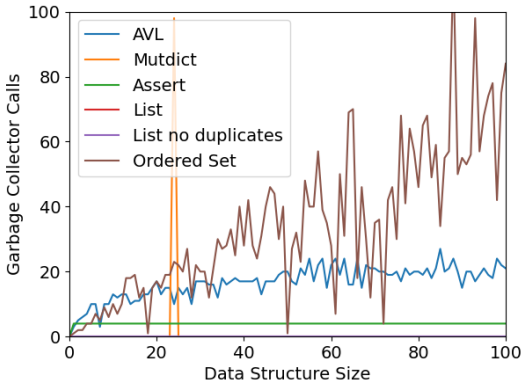


(b) Up to size 10 000.



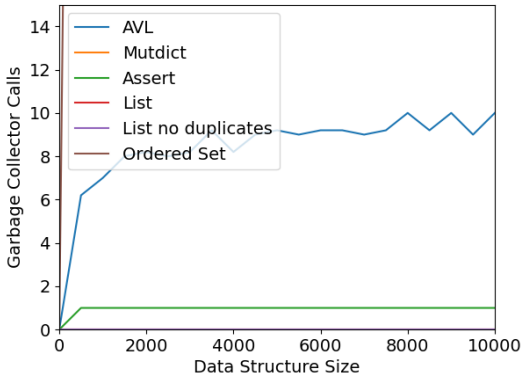
(c) Up to size 10 000. Y-Axis limited to 15

Figure 27: Garbage collector calls. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rancomplex described in section 5.3.



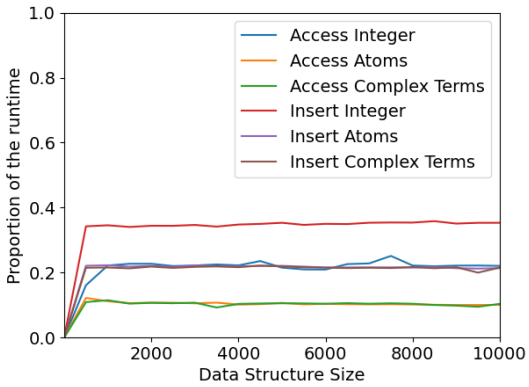
(a) Up to size 100.

(b) Up to size 10 000.



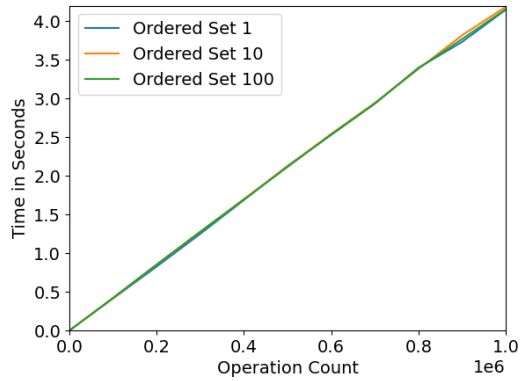
(c) Up to size 10 000. Y-Axis limited to 15

Figure 28: Garbage collector calls. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rancomplex described in section 5.3.

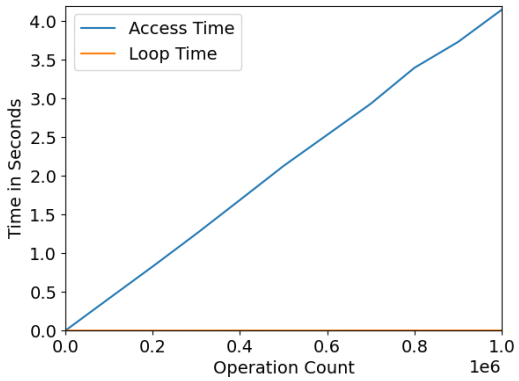


(a) Ordered Set.

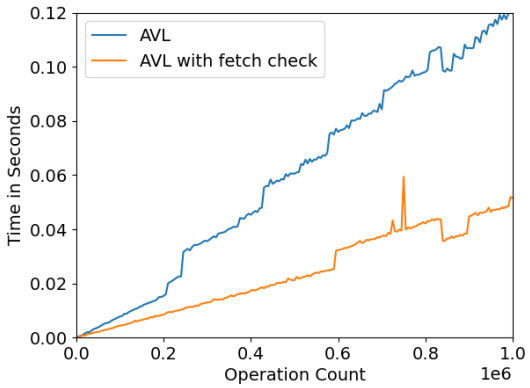
Figure 29: Garbage collection percent of total runtime. Constant operation count(1 000 000) paired with increasing data structure size.



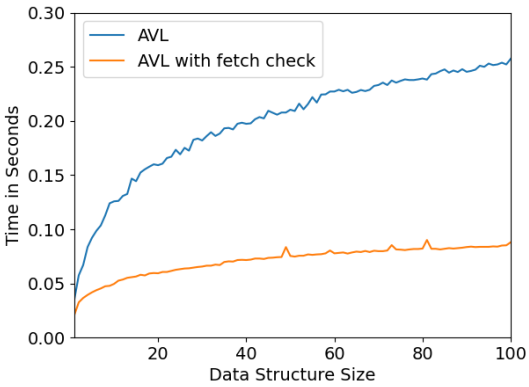
(a) Comparison of different number of access operation calls per recursion step. Increasing access operation count paired with constant data structure size (1000). The used data pattern is rannumbers described in Section 5.3.



(b) Comparison of the runtime of iterating through the access lists with performing the access operation. Increasing access operation count paired with constant data structure size (1000). The used data pattern is rannumbers described in Section 5.3.



(c) Comparison of the insertion operation when performed on an AVL Tree with and without fetch check. Increasing access operation count paired with constant data structure size (100 000). The used data pattern is rannumbers described in Section 5.3.



(d) Comparison of the insertion operation when performed on an AVL Tree with and without fetch check. Increasing data structure size paired with constant operation count (1 000 000). The used data pattern is rannumbers described in Section 5.3.

Figure 30: Miscellaneous other results.

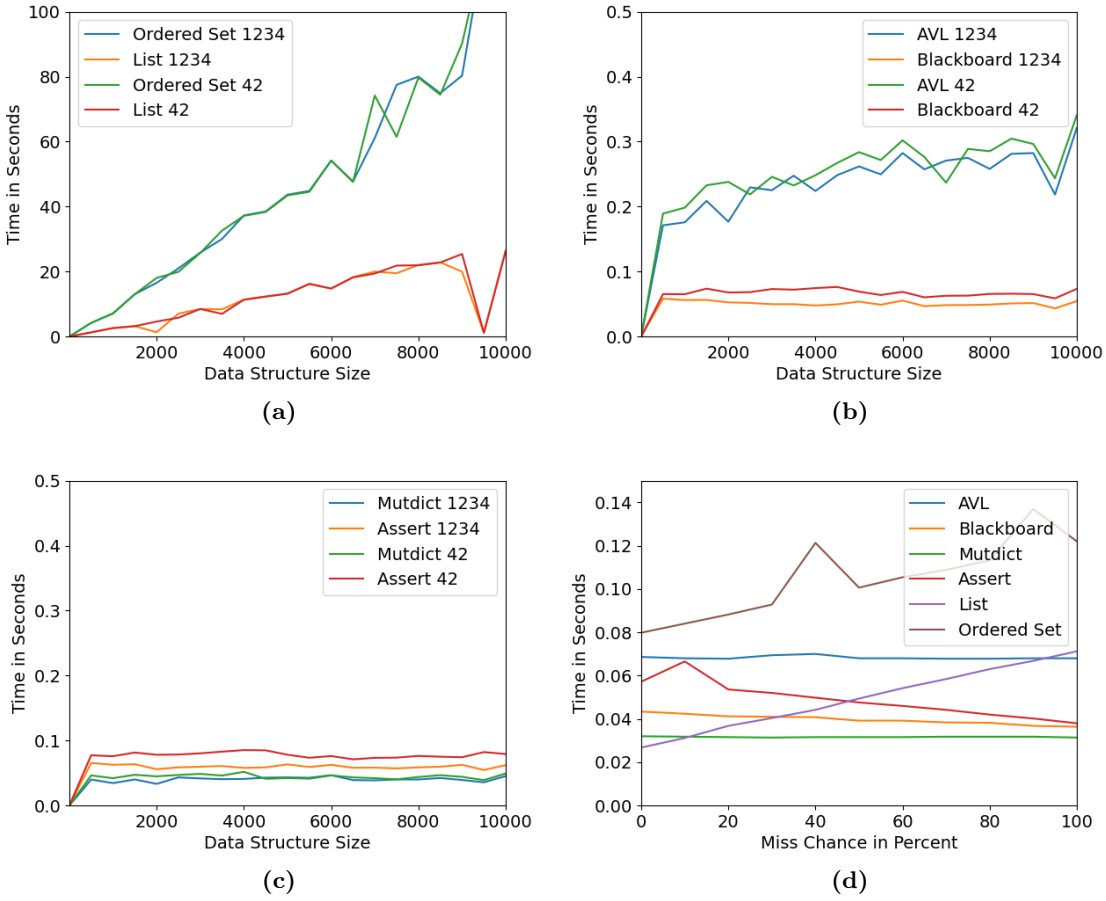


Figure 31: Comparisons of the randomnumber 42 and 1234. Displayed is the access operation with constant operation count 1 000 000 and atoms as keys. The used data pattern is ranatoms described in Section 5.3.

List of Figures

1 Access Operation with increasing data structure size paired with constant access operation count (1 000 000). The used data pattern is rannumbers described in section 5.3. 10
2 Access operation with increasing data structure size paired with constant access operation count (1 000 000). The used data pattern is ranatoms described in section 5.3. 11
3 Access operation with increasing data structure size paired with constant access operation count (1 000 000). The used data pattern is rancomplex described in section 5.3. 13
4 Access operation with increasing miss percentage paired with constant access operation count (1 000 000) and constant data structure size. The used data pattern is rannumbers10 described in section 5.3. 14
5 Access operation with increasing miss percentage paired with constant access operation count (1 000 000) and constant data structure size. The used data pattern is ranatoms described in section 5.3. 15
6 Access operation with increasing miss percentage paired with constant access operation count (1 000 000) and constant data structure size. The used data pattern is rancomplex described in section 5.3. 16
7 Access operation with increasing operation count paired with constant data structure size (10 000). The used data pattern is rannumbers described in section 5.3. 17
8 Access operation with increasing operation count paired with constant data structure size (10 000). The used data pattern is ranatoms described in section 5.3. 18
9 Access operation with increasing operation count paired with constant data structure size (10 000). The used data pattern is rancomplex described in section 5.3. 18
10 Insert operation with increasing data structure size paired with constant operation count (1 000 000). The used data pattern is rannumbers described in section 5.3. 19
11 Insert operation with increasing data structure size paired with constant operation count (1 000 000). The used data pattern is ranatoms described in section 5.3. 21
12 Insert operation with increasing data structure size paired with constant access operation count (1 000 000). The used data pattern is rancomplex described in section 5.3. 22
13 Insert operation with increasing operation count paired with constant data structure size (10 000). The used data pattern is rannumbers described in section 5.3. 23
14 Insert operation with increasing operation count paired with constant data structure size (10 000). The used data pattern is ranatoms described in section 5.3. 31

15 Insert operation with increasing operation count paired with constant data structure size (10 000). The used data pattern is rancomplex described in section 5.3. 31
16 Global stack bytes used. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rannumbers described in section 5.3. 32
17 Global stack bytes used. Insertion operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rannumbers described in section 5.3. 32
18 Global stack bytes used. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is ranatoms described in section 5.3. 33
19 Global stack bytes used. Insertion operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is ranatoms described in section 5.3. 33
20 Global stack bytes used. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rancomplex described in section 5.3. 34
21 Global stack bytes used. Insertion operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rancomplex described in section 5.3. 34
22 Heap bytes used by Assert and Blackboard. Constant operation count(1 000 000) paired with increasing data structure size. 35
23 Garbage collector calls. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rannumbers described in section 5.3. 36
24 Garbage collector calls. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rannumbers described in section 5.3. 37
25 Garbage collector calls. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is ranatoms described in section 5.3. 38
26 Garbage collector calls. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is ranatoms described in section 5.3. 39
27 Garbage collector calls. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rancomplex described in section 5.3. 40
28 Garbage collector calls. Access operation with constant operation count(1 000 000) paired with increasing data structure size. The used data pattern is rancomplex described in section 5.3. 41
29 Garbage collection percent of total runtime. Constant operation count(1 000 000) paired with increasing data structure size. 42
30 Miscellaneous other results. 43

31 Comparisons of the randomnumber 42 and 1234. Displayed is the access operation with constant operation count 1 000 000 and atoms as keys. The used data pattern is ranatoms described in Section 5.3. 44

List of Tables

List of Algorithms

List of Listings

1 Definition of a data structure. 6
2 Definition of a data pattern. 6

References

[ACHS88] Karen Appleby, Mats Carlsson, Seif Haridi, and Dan Sahlin. Garbarge collection for prolog based on wam. *Communications of the ACM*, 31(6):719–741, 1988.

[BM87] Jonas Barklund and Håkan Millroth. *Integrating complex data structures in Prolog*. Citeseer, 1987.

[Buc22] Iven Buchholz. Evaluation von datenstrukturen in prolog. Bachelor's thesis, Heinrich Heine University Düsseldorf, June 2022.

[CM12] Mats Carlsson and Per Mildner. Sicstus prolog—the first 25 years. *Theory and Practice of Logic Programming*, 12(1-2):35–66, 2012.

[FW86] Philip J Fleming and John J Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, 1986.

[KLB+22] Philipp Körner, Michael Leuschel, Joao Barbosa, Vitor Santos Costa, Verónica Dahl, Manuel V Hermenegildo, Jose F Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, et al. Fifty years of prolog and beyond. *Theory and Practice of Logic Programming*, 22(6):776–858, 2022.

[Sha97] Clifford A Shaffer. *A practical introduction to data structures and algorithm analysis*. Prentice Hall Upper Saddle River, NJ, 1997.