Runtime Evaluation of Prolog Data Structures

Philipp Körner, Christoph Ludolf, and Michael Leuschel

Heinrich Heine University Düsseldorf — Faculty of Mathematics and Natural Science — Department of Computer Science — 40225 Düsseldorf, Germany
{p.koerner,michael.leuschel}@hhu.de

Abstract. In Prolog, data structures are extremely simple: Ultimately, everything is a term, i.e., a functor and a number of arguments, which are terms themselves. This can be used to implement efficient data structures, and most Prolog standard libraries offer an implementation of AVL trees, association lists, etc. Recently, many Prolog systems also introduced data structures that are not term-based, such as blackboards or mutable dictionaries. The algorithmic complexity of operations on all these data structures is well-studied — yet, to our knowledge, no empirical comparison has been conducted for Prolog.
In this paper, we present our results on comparing different data structures. In particular, for each data structure, we benchmark the runtime of (i) an access operation (attempting to find a key or value), varying the percentage that the key or value is contained, (ii) an insert operation that adds a value or key-value pair to the data structure. Mutable data structures perform best, AVL trees are a solid alternative if a term-based data structure is required, and ordered sets perform the worst overall. The fact database can be a reasonable option to store data as well.

Keywords: Prolog · Data Structures · Evaluation

1 Introduction

Many Prolog systems [4] provide a variety of data structures, ranging from the ubiquitous lists, to ordered lists, binary trees, balanced binary AVL trees, and mutable dictionaries. In addition, one can assert and retract facts or use blackboard primitives to store data. Choosing an appropriate data structure for a task is an important decision regarding a program's performance. This decision may not come easy, as several factors influence the performance, e.g., what kind of data is stored (numbers, atoms or compound terms), the number of elements, in which patterns they are accessed, etc.

The theoretical asymptotic runtime of Prolog data structures is well understood. However, to the best of our knowledge, no empirical evaluation of how Prolog data structures perform in practice has been published.

With this paper, we provide a brief report on the results of some experiments we conducted. Due to page limitations, we cannot present all data we collected — possible combinations of aspects that can be imagined are too broad to provide a conclusive answer anyway. Instead, the intent is to raise awareness that

expectations on how well performing a solution actually is might differ from reality and to spark further discussion and research.

The motivation of this work stems from trying to improve the ProB animator and model checker [6,5]. It uses a variety of the above mentioned mechanisms: unordered lists for environments, ordered lists to collect used identifiers, AVL trees for large sets and relations, facts for the state space. It even stores some data like the model checking queue in C++. With the arrival of new mutable dictionaries in SICStus Prolog 4.7, we conducted an extensive empirical study to decide for which parts of ProB other data structures could be more appropriate.

The rest of the paper is structured as follows: we summarize the benchmarking methodology in Section 2 and present results in Section 3. Finally, we discuss conclusions and potential shortcomings of the benchmark sets in Section 4.

2 Methodology

This section describes the procedure used to collect the data. This includes a description of how the data structures are generated and how elements for the access and insertion operations are generated.

The benchmark code, raw data and additional plots, as well as the bachelor's thesis this article is based on [7], can inspected at:

https://github.com/pkoerner/prolog-runtime-data-companion

Benchmarks are executed in Prolog SICStus Prolog Version 4.7.1 [1] for Windows 10 x64 on an Intel i9-9900K CPU with 32 GB of DIMM 2400 MHz RAM. Runtime is measured by querying SICStus' `statistics` predicate for the current wall clock before and after each loop accessing or inserting the elements. In particular, all data structures (the base data and the elements that shall be accessed or inserted) are generated before the time is started. All benchmarks were executed sequentially. To minimize the influence of measurement inaccuracies, each benchmark is repeated five times and averaged using the geometric mean [2]. 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. Global data structures (i.e., facts and the blackboard) are cleared after each iteration.

In Table 1 and below, we will first give an overview of the data structures we considered. Then, we describe the data sets which are used for the benchmarks.

2.1 Observed Data Structures

For our experiments, focus on the data structures that are shipped with SICStus Prolog and its standard library. Below, we give a brief overview over the data structures and their expected runtime performance. We focus on using these data structures to represent `Key-Value` associations, assuming keys to be unique in the data structure. In our empirical analysis we will focus on two operations on these data structures: *insertion* of a `Key-Value` pair and *lookup* of the `Value` for a given `Key`.

Table 1: Overview of data types, expected runtimes and key takeaways from our experiments.

Data Type	Order / Index	Insert	Random Access	Key Takeaway
List	arbitrary	constant	linear, size/2 average case	use only for small lists
Deduped List	arbitrary	linear	linear, n/2 average case	prefer over ordsets
Ordered Sets	comparison	middle, linear, n/2 average case	linear, n/2 average case	use only if order useful (union,...)
AVL Trees	comparison	logarithmic average and worst case	logarithmic average and worst case	prefer if immutability is required
Mutdicts	term hash	constant, depending on key size	constant, depends on key size	very fast, use if appropriate
Mutarray	array index	constant	constant	like mutdicts, but only intkeys
Blackboard	undocumented*	undocumented†	undocumented	similar performance as fact data base
Facts	first argument functor	constant †	constant, but linear on collisions	similar performance as blackboard (cf. above)

Legend: *: probably key atom/integer; †: copies term, hence linear in size of term being inserted

Lists are singly linked finite sequences of unordered elements, potentially with duplicates. Thus, it is possible to insert an element in constant time (as it can be appended to the front). In contrast, to check whether an element is contained, in the worst case one needs to iterate over the entire list, resulting in a linear runtime linear in its length (but not in the size of distinct elements). In order to mimic an associative data structure, one can store terms in the form of, e.g., `Key-Value`; in our benchmarks however, we only store the key.

Deduped Lists are the same as the lists above, but with a different insertion predicate which checks whether the term is already contained. In the worst case, the entire list is scanned before an element is found or the insertion can occur, yielding a runtime linear in the size of its elements. As deduped lists use the same `member/2` predicate for accessing elements as lists, we do not benchmark this operation seperately.

Ordered Sets are provided by the library `ordsets`. Internally, they are ordered lists without duplicates. They enable linear implementations for set operations like intersection, union or difference. For both the access and insertion operation a runtime linear in the set size is expected on average.

AVL Trees (see, e.g., Section 6.2.3 of [3]) are self-balanced binary search trees. In Prolog, they are used for a tree implementation of "association lists". In theory, AVL Trees have a runtime that is logarithmic in the number of stored elements for both the access and insertion operation.

Mutdicts are unordered key-value collections, that are not term-based but use a hash table representation introduced in SICStus 4.7.0. Yet, they also support backtracking. The expected average runtime of both operations is constant in practice (in the absence of hash collisions). On rare occasions, update operations can take linear time.

Mutarrays are a SICStus Prolog implementation of dense arrays, also added in SICStus 4.7.0. They are a special case of Mutdicts and provide a mapping from an integer to an arbitrary value. Because arrays throw an exception if an index out of bounds is accessed, one cannot access indices that are *not* contained (too large, or negative). We expect both insertion and random access to run in constant time in practice. Note that even though Mutdicts and Mutarryas are mutable, the operations on them can still be backtracked.

Blackboards are a per-module repository to store elements as key-value pairs. Keys must be integer or atoms. It is expected that they perform well for both the insertion and access operation. The likely runtime of both operations is constant in the size of the data structure and linear in the key size.

Facts can be added dynamically to the Prolog clause database using `assert/1`. The insertion operation is expected to perform in constant time regardless of the amount of stored elements, yet linear in the size of the stored key-value pair (as it is copied). The access operation, may vary depending on the key type. This is because facts are indexed on the functor of the first argument (i.e., the functor of the key, as we assert a term `mydb(Key, Value)`). A constant (wrt. the size of the fact data base) runtime is likely for the insert operation. For the access such a fact with atoms or integer values as keys, a constant runtime is also predicted. When compound terms are used as keys, the runtime may be linear in the number of facts, depending on how often the functor of the keys is re-used.

2.2 Data Generation

In our benchmarks, we generate random keys to (i) construct, (ii) access or (iii) insert into the data structures. Values, on the other hand, will always be the atom `true`. Below, we describe how the different types, or rather patterns, of key types are generated.

rannumbers is used to generate integer values as keys. The data structure of size N will be generated by inserting a random order of the numbers $1, 2, \ldots, N$. In order to generate a key that is definitely known, numbers are drawn randomly from the same interval. If keys are required that are not contained in the data structure, reasonable values are -1 oder $N + 1$.

rannumbers10 is similar to the **rannumbers** pattern. However, the base data only contains all multiples of 10, i.e. the sequence $10, 20, N * 10$. This allows benchmarking accessing keys that are *not* contained in the data structure, yet *would* be placed somewhere "in the middle". Uncontained keys are generated by drawing a number that is contained and randomly subtracting a value from 1 to 9.

ranatoms is used to generate atoms as keys. To generate N atoms, we first draw an integer from $1, \ldots, N$ (a seed), and, second, use this integer to generate two random ASCII characters c_1 and c_2. The generated atom will then have the shape: `myatom_`c_1c_2`seed`. To generate atom that are not contained, we use the same idea, but employ $N + 1$ or $N * 2$ as seeds.

rancompound is used to generate compound terms for keys. Just like atoms, N compound terms are generated by drawing an integer from $1, \ldots, N$ (a seed). The seed is used to generate two ASCII values i_1 and i_2, which correspond to characters c_1 and c_2. Ultimately, the term will have the shape: `term_`c_1`(term_`c_2`(seed,`i_2`,`i_1`))`. Terms that are not contained in this set are generated by using either $N + 1$ or $N * 2$ as the seed.

3 Results

In this section, we summarize the data collected from the benchmarks. We will first focus on a random access operation, and investigate insertions afterwards.

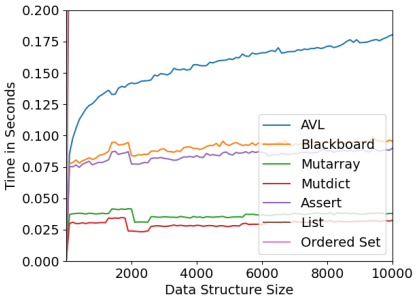
3.1 Random Access

In order to benchmark the (random) access operation, we instantiate (a) a list of elements that will be inserted into the data structure, (b) a list of elements that we will access, and, optionally, (c) a list of elements that are not contained that we try to access. We measure many iterations of the same operation in one benchmark to avoid errors due to clock resolution. The list of elements (b) are generated randomly as described in Section 2.2 in order to avoid, e.g., always accessing the first element in the list. In theory, this will let us compare the average case between different data structures.

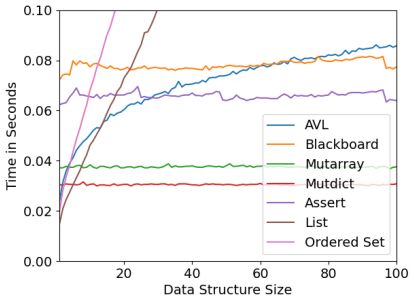
Below, we will consider (i) integers as keys, (ii) atoms and compound terms as keys and (iii) the influence of attempting to lookup keys that are not contained.

Integer Keys In our first experiment, we used integer values as keys. The runtimes of 1 000 000 accesses for different data structures in relation with the size of the data structure can be seen in Fig. 1.

Results As expected, lists and ordered sets perform linearly. However, for very small data structures (less than 15 elements), lists perform better than AVL trees. For even smaller sizes (less than 5 elements), they even beat mutable data structures. Up to size 25, AVL trees seem to perform better than facts.



(a) Up to data structure size 10 000. The plots for lists and ordered sets are hard to distinguish near the Y-axis.



(b) Close-up of data structure sizes 0–100.

Fig. 1: Runtime of 1 000 000 random accesses depending on the data structure size.

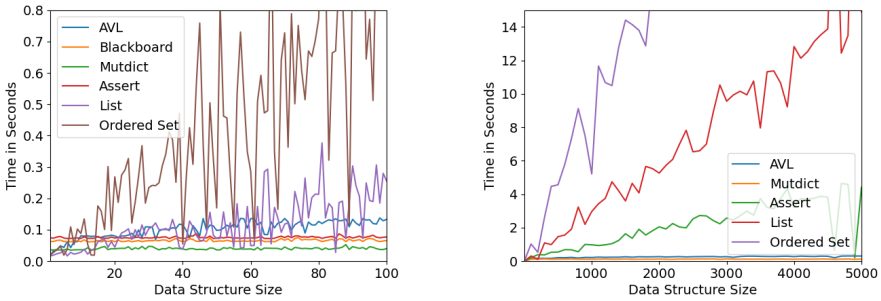
Surprises Ordered sets have, on average, a worse runtime than lists. This is due to the extra comparisons that are made while traversing the internal list. The overhead here, however, is more significant than we expected. We also did not expect that (a) the mutable data structures are more than 2× faster than accessing dynamic facts. The mutable array seems to be outperformed by the mutable dictionary. Further, the blackboard primitive is even slower than the dynamic database.

Atoms and Compound Terms as Keys Next, we investigated what influence the data type of the key has; first, by using atoms and, second, by employing compound terms. The results are depicted in Fig. 2a and Fig. 2b, respectively. Note that mutable arrays only support integer indices, and the blackboard primitive does not allow compound terms as keys. Thus, we could not include them in the results.

Results Using non-integer keys made the runtime vary significantly. However, the overall trend remains, that ordered sets are the slowest data structure, followed by the list. On the other hand, the mutable dictionary is still the fastest. We can see, however, that the blackboard is slightly faster than the dynamic database if the keys are atoms.

Surprises Once compound terms are used as keys, however, AVL trees are significantly faster than the dynamic database. This may be due to the low variance in the generated functors used for indexing. However, in typical applications, names for functors are very limited as well.

Varying the Miss Chance So far, we have considered only successful lookups. In Fig. 3, we now consider the runtime for a fixed-size data structure in relation to



(a) Atoms as keys.

(b) Compound terms as keys.

Fig. 2: Runtime of 1 000 000 *access* operations in relation to the data structure size.

how many *unsuccessful* lookup attempts are made. This experiment uses random integers again, in particular the `rannumbers10` pattern. Figs. 3a and 3b show the same benchmark of size 10 000, but as the runtimes for lists and ordered sets are in different orders of magnitude, they are shown separately.

Results As the miss chance increases, lists become slower. This is expected, as for every missed item, the entire list must be traversed. All other data structures, however, are getting faster or remain unaffected. There seems to be a sweet spot between 40–60% miss rate where ordered sets and lists produce the same runtime on average.

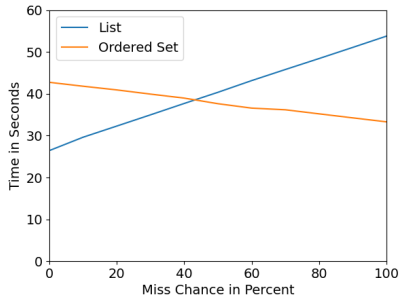
Again, the mutable dictionary outshines all other data structures. The dynamic database is faster than AVL only for larger data structures, and its runtime is only little influenced by the miss chance.

Surprises Ordered sets perform worse than expected here. The miss chance must be pretty high for them to even beat lists. In any case, using any other data structure does no harm, even for small sizes.

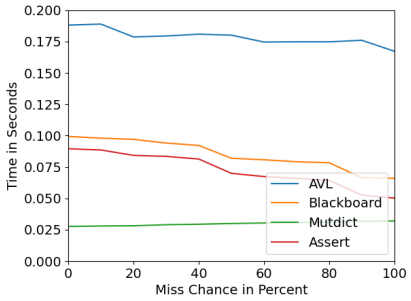
Atoms and Complex Terms as Keys Due to page limitations, we omit corresponding figures. For these types, ordered sets perform even worse and were unable to beat lists when the size was 10 000. They seem to be more efficient with a very high miss chance for data size 10 when using compound terms, and data size 15 when using atoms as keys — but not vice versa.

3.2 Insertion

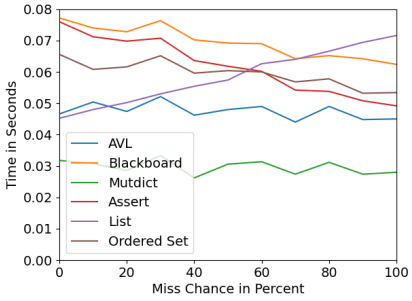
The insert benchmark is performed similar to the access benchmark. First, a data structure x with N elements is generated. Note that this might not be in a specific order. Then, a list of elements to insert in this data structure is generated



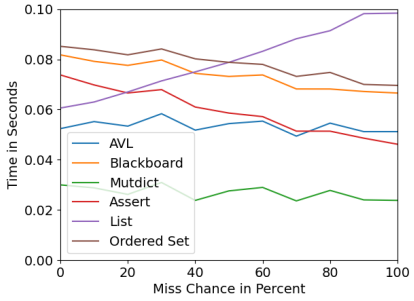
(a) Data structure size 10 000



(b) Data structure size 10 000



(c) Data structure size 10.



(d) Data structure size 15.

Fig. 3: Runtime of 1 000 000 *access* operations in relation to the miss chance for some data structure sizes.

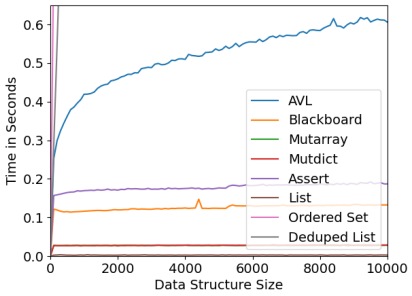
— these might be already contained or not. The resulting data structure is discarded (unless mutable) and further insertions are repeated on x.

Below, we will consider (i) integers as keys, (ii) atoms and compound terms as keys and (iii) the influence of a fetch check for AVL trees.

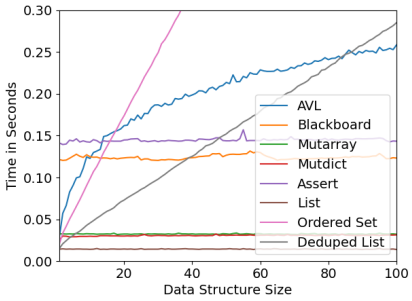
Integer Keys As before, we will first start with integer keys. The results can be seen in Fig. 4.

Results As expected, a list insertion that does not check for duplicates is the fastest operation so that the plot is hard to distinguish from the X-axis. There seems to be no difference between inserting in a mutable dictionary and in a mutable array. The blackboard performs better than the dynamic database. AVL trees get slower with size. To no surprise, lists that check for duplicates and ordered sets are quickly off the chart.

Surprises Ordered sets perform worse than iterating over the entire list to scan for duplicates. Mutable data structures perform about five times better than



(a) Up to data structure size 10 000. The plots for mutdicts and mutarrays overlap. The plot for lists is hard to distinguish from the X-axis, the ones for ordered sets and deduped lists are close to the Y-axis.



(b) Close-up of data structure sizes 0–100. The plots for mutdicts and mutarrays overlap.

Fig. 4: Runtime of 1 000 000 random *insertions* depending on the data structure size.

the dynamic database. This is surprising as we expected that the overhead for indexing and hashing to be similar.

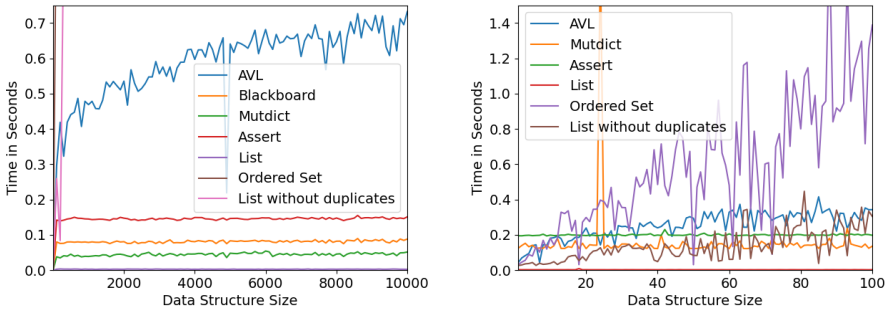
Atoms and Compound Terms as Keys We are again interested in the influence of the data type of the key on the runtime. We will once again use atoms (Fig. 5a) and compound terms (Fig. 5b). As before, we do not include data structures that do not support those key types.

Results The overall results are similar to the ones for integer keys. Yet, for small data structures, in particular for regarding compound terms, the picture is less clear. There, it seems to depend more on the concrete actual values. However, the mutable dictionary is still a safe bet.

Surprises There is no further upset here.

AVL Fetch Check In further experiments, we were surprised that inserting into the AVL tree seems to perform differently depending on whether data was already contained or not. Thus, we experimented with adding a "fetch check" to the AVL, as depicted in Listing 1.1: instead of just storing the element, we first perform a fetch and check, whether the resulting element is already stored. If so, we can avoid storing the value, and need to copy it only otherwise. In other words, we compare the lookup of a value with a full write in this particular experiment.

In this experiment, we tried to store key-value pairs that were already contained. The results are depicted in Fig. 6.



(a) Atoms as keys. (b) Compound terms as keys.

Fig. 5: Runtime of 1 000 000 *insert* operations in relation to the data structure size.

Listing 1.1: AVL store with fetch check predicate

```
avl_store_fetch_check([],A,A).
avl_store_fetch_check([H|T],In,Out) :-
  (avl_fetch(H,In,true) -> In2=In ; avl_store(H,In,true,In2)),
  avl_store_fetch_check(T,In2,Out).
```

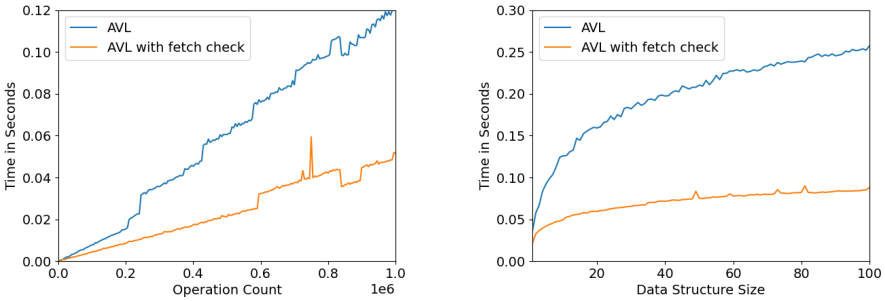
Results In Fig. 6a, we consider an AVL tree with a fixed size of 100 000 elements and depict the runtime in relation to the number of insert operations. Since the tree does not increase in size, we can observe a rough linear increase in runtime depending on the number of accesses. Replacing the term seems to cost more than double than just accessing it.

We vary the data structure size but keep the operation count fixed in the experiment whose results we show in Fig. 6b. Thus, we see the typical logarithmic curve for AVL trees. It seems like the fetch check is more important the larger the data structure.

4 Discussion

In this article, we shared some of our experiments regarding the runtime performance of Prolog data structures. Some of them also served as a stress test, discovering a segmentation fault in earlier versions of SICStus' mutable data structures.

Conclusions Due to the sheer number of possible access patterns and other factors that might influence the choice of an appropriate data structure, it is hard to draw conclusions that are universally valid. Regardless, we try our best:



(a) Runtime in relation to the operation count for an AVL tree of size 100 000.

(b) Runtime of 1 000 000 insertions in relation to the size of the AVL tree.

Fig. 6: Comparison of the performance of *insertions* into AVL trees with and without a fetch check. The key-value pair to be stored is already contained.

- use data structures that are not term-based, such as mutdicts or mutarrays, if available and appropriate;
- ordered sets can be surprisingly slow due to extra term comparisons — avoid unless you depend heavily on the ordering (e.g., to obtain linear complexity for set operations like union, intersection and difference);
- lists are very performant if sufficiently small (about less than 15 elements);
- if you often overwrite data with itself in an AVL tree, adding a lookup makes you faster;
- AVL trees make a good all-round default: they are immutable, sufficiently fast, but their terms usually are not easily readable;
- operations that copy terms can be a serious efficiency bottleneck, in particular inserting into a blackboard and asserting to the fact database;
- as blackboards are scoped to an entire module and only allow atoms and numbers as keys, they can be very inconvenient if several data structures need to be maintained. In particular, one cannot easily instantiate several copies, and instead has to create keys with specific prefixes using, e.g., `atom_concat`.

4.1 Future Work

The conclusions outlined above are preliminary and will ultimately not suffice to pick the most appropriate data structure. More data is needed, in particular on:

More Operations In this paper, we focused on lookup and insertion operations. However, there are more operators that are often used, e.g., the initial construction of the data structure, update and removal of data, as well as the union of two data sets (e.g., `concat`). Additionally, we did not consider how the data structures behave during backtracking. Further insights on the runtime of these operations are ultimately required for an informed selection of a data structure.

Better Generators While our data generators uncovered differences between different data types, such as numbers, atoms and compound terms, the generated values are not really realistic. Typically, they have significantly more variance (in case of names) or less variance (in case of term functors). Terms might also be significantly deeper nested and larger in the argument count. All this may influence the runtime needed to make comparisons and to calculate hashes.

More Benchmarks There are more real-world scenarios that might be worthwhile to investigate: access often is not randomly but sequential instead (just getting all data, converting an AVL tree to lists, ...). We also attempted to collect data on used memory; however, the results fluctuate too heavily. Here, one would require better insight on the allocated memory from the Prolog system.

More Prolog Implementations and Data Structures So far, we have only run our experiments with SICStus Prolog. Other systems offer different standard libraries or slightly different versions of the data structures. Further, SWI-Prolog offers an implementation of dictionaries [8], B-Prolog implements hashtables and various systems implement some form of arrays. Thus, our results need to be validated for other Prolog systems as well. It would also be interesting to investigate more tree-based data structures.

Detailed Investigation of Outliers For some data structure sizes, we noticed weird outliers, both positive and negative. However, we were able to reproduce them consistently — even with different random seeds. We want to investigate those in more detail to gain a better understanding what is happening for these data points.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Carlsson, M., Mildner, P.: SICStus Prolog—the first 25 years. Theory and Practice of Logic Programming **12**, 35–66 (2012)
2. Fleming, P.J., Wallace, J.J.: How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results. Communications of the ACM **29**(3), 218–221 (1986)
3. Knuth, D.: The Art of Computer Programming, Volume 3. Addison-Wesley (1983)
4. Körner, P., Leuschel, M., Barbosa, J., Costa, V.S., Dahl, V., Hermenegildo, M.V., Morales, J.F., Wielemaker, J., Diaz, D., Abreu, S., Ciatto, G.: Fifty Years of Prolog and Beyond. Theory and Practice of Logic Programming pp. 1–83 (2022)
5. Leuschel, M.: ProB: Harnessing the Power of Prolog to Bring Formal Models and Mathematics to Life, LNAI, vol. 13900, pp. 239–247. Springer (2023)
6. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Software Tools for Technology Transfer **10**(2), 185–203 (2008)
7. Ludolf, C.: Runtime Evaluation of Data Structures in Prolog. Bachelor's thesis, Heinrich-Heine-Universität Düsseldorf (2023)

8. Wielemaker, J.: SWI-Prolog Version 7 Extensions. In: Proceedings WLPE (Workshop on Logic-based Methods in Programming Environments). Aachener Informatik-Berichte, vol. 2014-09, pp. 109–124. RWTH Aachen (2014)