The Split List

Bruno Carneiro, **Carlos Ramos** and **Victor Pinheiro**
Institute of Computer Science, University of Tartu

UNIVERSITY OF TARTU 1632

Introduction

We present the **Split List** Go library, a high-performance implementation of a novel data structure heavily inspired by the Skip List's [4] probabilistic nature, and the B-Tree[1]. The Split List is a compressed Skip List, with worst-case space complexity $O(n)$, a significant improvement over the Skip List's $O(n \cdot \log n)$, while at the same time being possibly more efficient.

From Skip List to Split List

The Split List can be visualized in Figure 2. The left part shows the geometric $\log n$ height hierarchy, inherited from the Skip List. To the right we can see the internal structure of each height level, alongside the insertion mechanism.

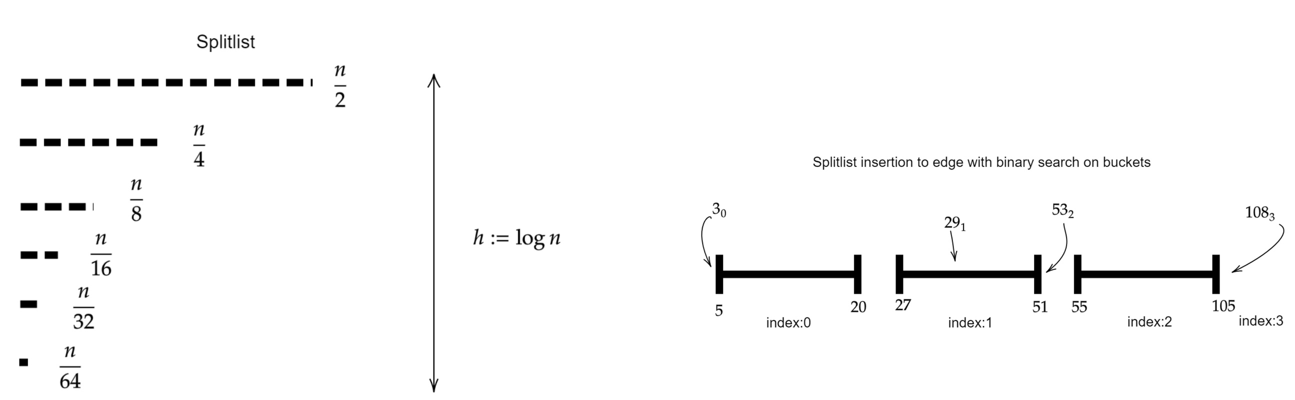

Fig. 2: Split List general architecture and edge architecture.

Lookups and Concurrency amenability

Searching in the B-Tree is done almost as in every other tree, first by recursively going down the tree, to then using some find operation, such as binary search, in a node. The Skip List uses a traversal algorithm using the shortest path from the highest point of the Skip List until the finding query point, or it "falls off." The Split List, however, uses nothing but binary searches. Starting from the highest level, which in the Skip List would be the one with the least amount of data, while in the Split List it has the most, a binary search happens over the ordered sub arrays' maximum values, and then once again in the chosen bucket. It is trivial to see that this much less-specific method would also imply not having a global lock, similarly to the skip-list, while at the same time benefiting from having a more cache-locality-friendly structure, like the B-Tree.

Benchmarks

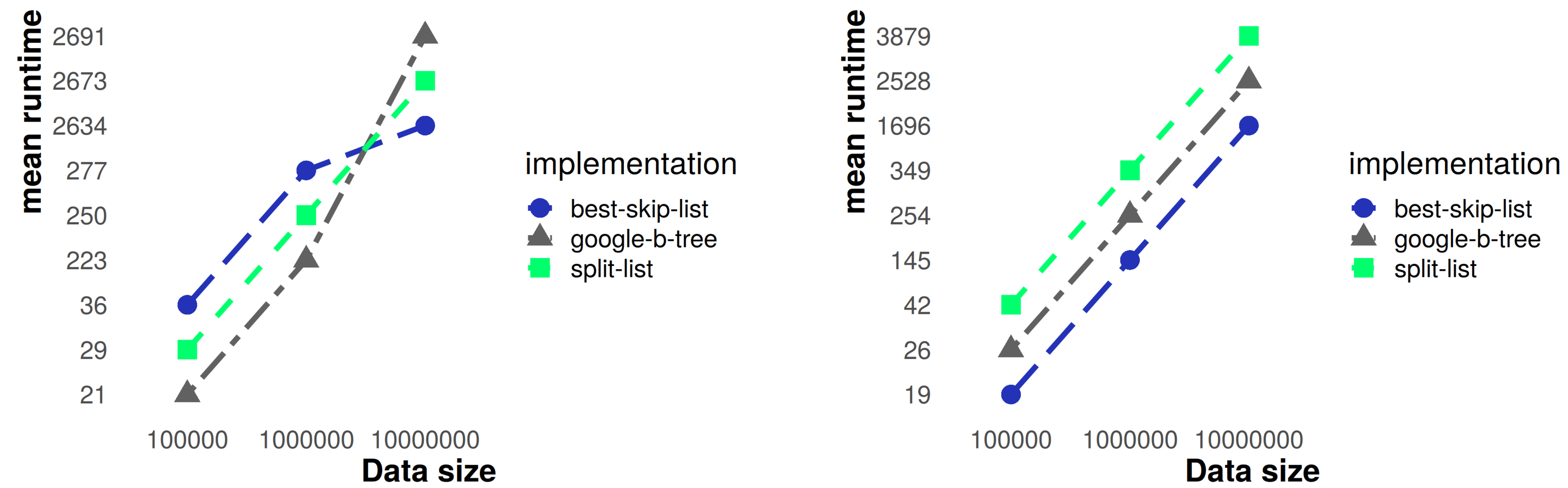

Fig. 3: Insert and lookup benchmarks.

In the graphs above we can see a comparison of google's state-of-the-art implementation of a B-Tree[2], the best skip-list implementation we could find, as seen on a survey[3], and our 'SplitList'. To the left we have a benchmark that measures the average time, over millions of runs, to insert 100 thousand, 1 million and 10 million integers, and to the right lookups. Both benchmarks insert integers sequentially. It is clear that all of them are very close, however, there's some noticeable overhead in the Split List.

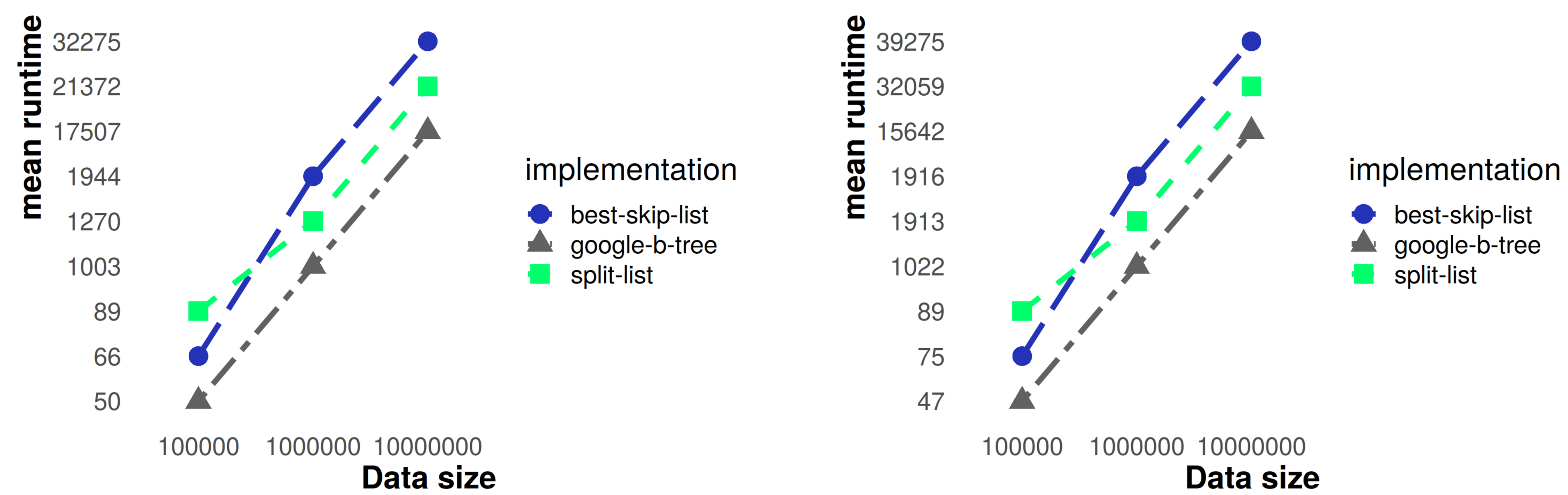

Fig. 4: Lookup when element is not in the list.

The next benchmarks are done in the same way, save for the fact that the integers are now inserted at random, with the distribution being uniform, but the order changing, in every single iteration. While in the sequential benchmarks sometimes the skip list would best the split list, in the these this situation never happens, further evidencing that it indeed satisfies the requirements for being a compressed skip list, that is, retaining the same performance while having a smaller space complexity.

Rank and Select

Furthermore, we implement two efficient methods for the *rank* and *select* operations. An optimal selection for sorted sequences would require to iterate over items that cannot be smaller than the desired input k. The removal and iteration would require $O(log(\frac{n}{k}))$ operations. Instead, we use a caching mechanism that updates the items to be selected. This leads to keep track of the moment of cache invalidation and update. In case of an update, the elements are kept in a single flat list, sorted. Thus, the amortized time for accessing on every selection is reduced to a constant factor $O(k)$. The same mechanism is applied to *rank*, with the difference that we rely on a bisection over the collection of times in order to find the exact position of the rank.

Complexities

Let k be the size of the bucket in Split List and n be the size of the list. The worst case complexities of the described data structures are represented in the following table:

Data structure/ Op.	Insertion	Lookup	Delete	Space
Split List	$\mathcal{O}(k)$	$\mathcal{O}(\log n + \log k)$	$\mathcal{O}(k)$	$\mathcal{O}(n)$
Skip List	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n \cdot \log n)$
B Tree	$\mathcal{O}(logn)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

For Split List, considering that by default order $k = 1024$, then the average case of insertion is very fast. Search tends to be slow only when we have to search from more than one edge but the elements are usually on the first edge thanks to the probabilistic nature of the list. In that case, the time complexity will be $\mathcal{O}(\log n)$ for search.

Conclusion

In sum, we managed to successfully meet the original objectives of the project, to create a high-performance implementation of the Skip List. Our Implementation beats all current Skip List implementations in Go by considerable margins, while at the same time being roughly twice as heap-memory effective, living up to the claim of it being a compressed Skip List, to the point of it being almost as fast as google's own implementation of a B-Tree.

Acknowledgements

The authors are grateful for the support by StudyIT, European Regional Development Fund, the Archimedes Foundation, and the University of Tartu.

References

GitHub Repository: https://github.com/brurucy/splitlist

[1] Douglas Comer. "Ubiquitous B-Tree". In: *ACM Comput. Surv.* 11.2 (June 1979), pp. 121–137.

[2] Google. *Google's B-Tree.* `https://github.com/google/btree`. 2014.

[3] MauriceGit. *Survey of Skip List Implementations.* `https://github.com/MauriceGit/skiplist-survey`. 2018.

[4] William Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees". In: (1990).