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.

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.

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 ‘Split List’. 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.

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, while the order changing, in every single iteration. While in the sequential benchmarks sometimes the skip list would best the split list, in 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.

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 $O(\log(n))$ 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 $O(1)$. 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.

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

References