Advanced Data Structures: Final Report

Steil, Patrick ✉
2513240

Abstract

In this work, different data structures are implemented that can efficiently answer two types of queries: predecessor and range-minimum queries. For this purpose, the Elias-Fano coding and several "Block-Lookup" data structures are implemented and tested. The project is part of the "Advanced Data Structures" lecture at KIT in the summer semester 2023 with Dr. Florian Kurpicz.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Predecessor, Range Minimum Queries

Supplementary Material *Software (Code)*: `https://github.com/PatrickSteil/BitDatastructures`

1 Algorithms and Data Structures (DS)

Predecessor Data Structures

Predecessor data structures are unique data structures developed to efficiently solve the problem of predecessor access. The predecessor access problem is to find the predecessor (the element with the highest value that is less than a given element) in a set of elements. Without additional memory consumption, elements can be stored sorted or in a binary search tree, whereby a predecessor query can subsequently be answered with a runtime of $\mathcal{O}(\log(n))$. However, this is too slow in practice, which is why techniques such as *x/y-fast-tries* [3] or *Elias-Fano encoding* [1] have been developed. The idea of the last technique is to divide the representation of numbers into two parts, on the one hand, the *most significant* bits with a length of $\lceil \log(n) \rceil$ and on the other hand the remaining bits. This way, all numbers with the same *most significant* bits are "grouped" during a query, meaning a query only has to look at one "group" of numbers. However, this may result in all numbers being scanned if the numbers are not favourable.

Range-Minimum Queries

Range minimum queries (RMQ) refer to a particular type of search operation in data structures where the minimum value within a given interval (range) of elements in a set is to be found. The goal is to efficiently find the biggest value smaller than a given element among a subset of elements. Analogous to the predecessor data structures, trade-offs exist between additional memory and runtime (the different pre-computation times can be neglected for static data). To answer RMQ quickly, one can use $\Theta\left(n^2\right)$ additional memory to calculate all possible queries in advance and thus obtain a constant query time (only a memory access). One can now only precompute queries of specific ranges to reduce the impractical quadratic memory. If the size of this range is cleverly chosen, one obtains constant query time and down to only linear additional memory.

2 Implementation

Elias-Fano

For an efficient Elias-Fano implementation, one needs a bit-vector data structure that can efficiently answer SELECT query. The current Elias-Fano data structure consists of two bit

vectors: *upperBits* and *lowerBits*. The former encodes in $2n + 1$ bits the *most significant bits* (MSB) of the numbers, and in *lowerBits*, the remaining bits (LSB) of the numbers are stored straightforwardly. Please see Appendix A on the "minor" Elias-Fano Problem, which I mentioned in person. Otherwise, my implementation of PREDECESSOR is based on the description of [2]. To identify the predecessor of the number x, I use SELECT queries to find the "leftmost" and "rightmost" index of numbers for which the MSB is equal to the MSB of x. Subsequently, scanning this range to find the predecessor is straightforward.

Range-Mimimum Queries

For the range-minimum queries, I implemented all 3 data structures, i.e., the one with $\mathcal{O}\left(n^2\right)$, $\mathcal{O}\left(n \log\left(n\right)\right)$ and the one with $\mathcal{O}\left(n\right)$ additional memory. I have not implemented the latter data structure precisely as in the lecture; namely, I have not implemented the *Cartesian Trees*. Instead, I have split the query into three parts. I check which parts of the given range are "covered" by blocks and then scan the initial and final part (from the left index to the first "completely covered" block and analogously from the last covered block to the end) and the covered blocks. The initial and final check is usually done with Cartesian trees; I have implemented a linear search over the part of the array. Note that this approach is very cache-efficient and for small enough ranges faster than using Cartesian trees.

3 Experiments

The programming language used was `C++`, and the compiler was `G++` with optimisations enabled. All experiments were conducted on a ThinkCentre running Ubuntu 22.10 with an `Intel(R) Core(TM) i5-8500T CPU` @ 2.10GHz processor.

Elias-Fano

The performance (based on the provided benchmarks) of the Elias-Fano encoding can be seen in Figure 1. It can be noticed that both the memory consumption and the query depend on the distribution of the numbers. In the first file, 1 000 000 numbers are present in the *total* range of $\left[0, 2^{64}\right)$, i.e., scattered throughout the universe. On the other hand, in the file *File 2*, only "small" numbers are present, i.e., very one-sidedly distributed in the universe. If one would not resize the universe to the maximum number occuring in the given set, basically all MSB are 0 for all numbers, resulting in no speedup compared to linear search.

Range-Minimum Queries

Figure 2 visualises the performance of the RMQ data structures, namely the two with $\mathcal{O}\left(n \log n\right)$ and $\mathcal{O}\left(n\right)$ additional memory consumption. The data structure with $\mathcal{O}\left(n^2\right)$ memory consumption could not be executed due to too high memory consumption. It can be seen that the memory consumption between the two DSs is very different, namely, as expected, with significantly less memory for the DS with linear memory overhead. Interestingly, the query performance varies significantly between the DS with linear memory overhead because the ranges in the second file are relatively small. In my implementation, this only calls a local sequential search, which is the most cache-efficient and, for small ranges, the fastest way. Otherwise, the query performance of the $\mathcal{O}\left(n \log n\right)$ DS does not depend on the range since the query performs lookups "independent" of the given range.

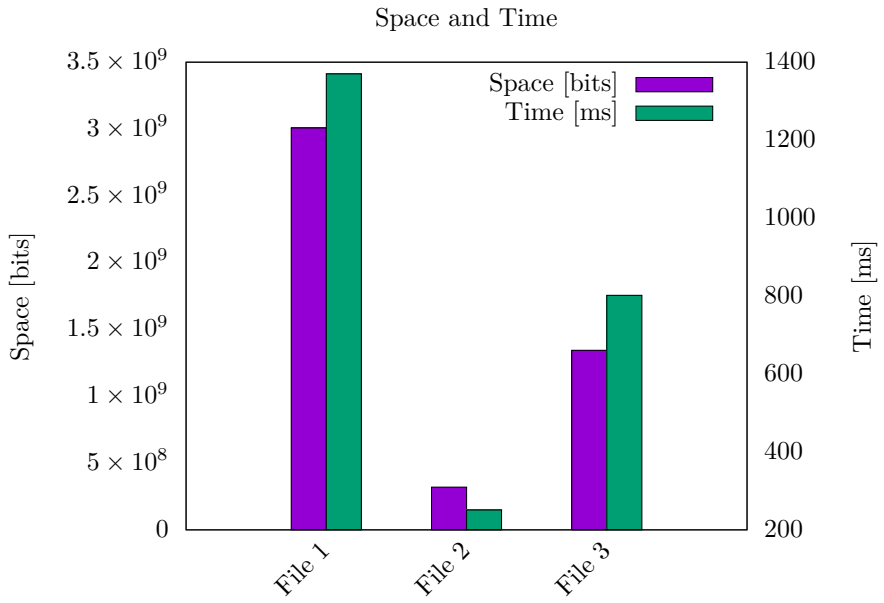

Figure 1 Space consumption and query time (for 1 000 000 predecessor queries) for my implementation of Elias-Fano. The three files are the three provided benchmark files.

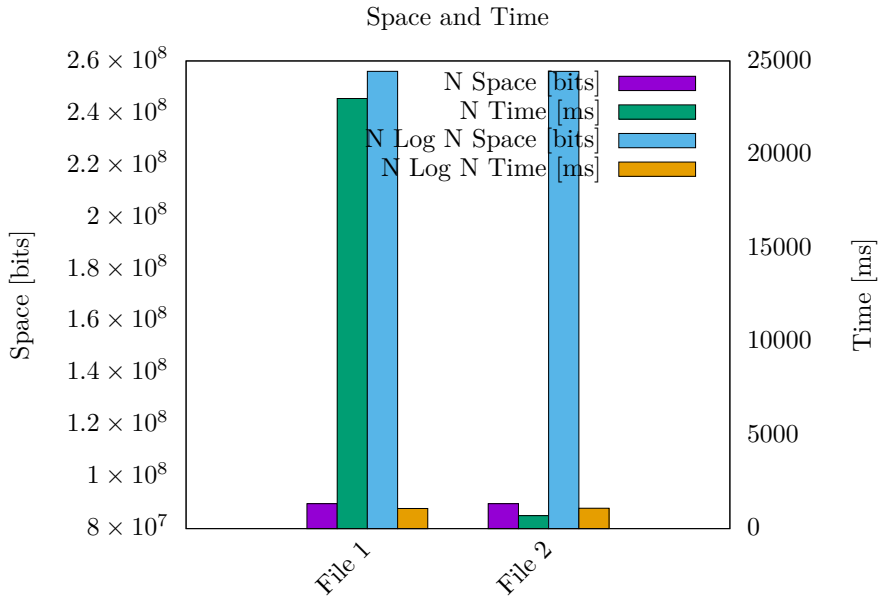

Figure 2 Shown is the memory consumption and performance of the range-minimum query data structures *N Log N* and *N* on the benchmark files. Note: the $\mathcal{O}\left(n^2\right)$ data structure could not be executed on the benchmarks due to high memory consumption.

References

1 Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, apr 1974. doi:10.1145/321812.321820.

2 Giulio Ermanno Pibiri and Rossano Venturini. Succinct dynamic ordered sets with random access, 2020. arXiv:2003.11835.

3 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983. URL: https://www.sciencedirect.com/science/article/pii/0020019083900753, doi:https://doi.org/10.1016/0020-0190(83)90075-3.

A A Note on Elias-Fano

In the script, the first $\lceil \log(n) \rceil$ bits were defined as MSB, but this definition caused problems with e.g., the following numbers:

$$0, 1, 10, 13, 32, 64, 1\,926\,720\,561\,250\,547\,757, 4\,354\,482\,840\,745\,948\,438, 18\,446\,013\,172\,173\,323\,708$$

Now, looking at the various MSBs of length $\lceil \log(n) \rceil = 4$, we see:

$$(0000)_2, (0001)_2, (0011)_2, (1111)_2 \tag{1}$$

If *upperBits* has a length of $2 \cdot 9 + 1 = 19$, then to decode the last number, you would need to acces the index $(1111)_2 + 8 = 23$, which is out of bounds for *upperBits*. Therefore, I set the length of the *upperBits* to $3 \cdot n$, as discussed in the lecture.