Advanced Data Structures: Final Report

Atalay Donat
2312438

Abstract

In this project, one predecessor (with Elias Fano) and three range minimum query data structures are implemented. This paper describes the algorithms, explains some implementation details and shows some evaluation results.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Predecessor, Range Minimum Queries

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Two queries that are often used when working with lists are predecessor quries and range minimum queries. The operations are defined as follows on a list v: $RMQ(s,e) = \arg\min_{s \leq i \leq e} v[i]$ and $pred(x) = \max\{y \in v : y \geq x\}$. In Sec. 2, algorithms and data structures for solving these queries are described. Some important implementation details are explained in Sec. 3. Finally, experimental evaluation on these algorithms and data structures are shown in Sec. 4.

2 Algorithms and Data Structures

2.1 Predecessor

For answering predecessor queries, Elias Fano Coding is used, which given an array containing n distinct integers from a universe $U = [0, u)$, represents the array using $(2 + log\left\lceil\frac{u}{n}\right\rceil)n$ bits while allowing $O(log\frac{u}{n})$ predecessor time.

2.2 Range Minimum Query (RMQ)

Three data structures that can be used to answer range minimum queries are described in the following. They all support $O(1)$ query time but differ in their build time and space complexity.

2.2.1 (Naive) Quadratic RMQ

One naive way of solving RMQs is to store the solutions of all possible ranged beforehand and then accessing them in each query. This uses $O(n^2)$ space, as there are $O(n^2)$ possible ranges in a list of size n.

2.2.2 Loglinear RMQ

One improvement that can be made to the naive implementation is to store solutions only for intervals of length 2^k for every k. This reduces the space complexity to $O(nlogn)$.

2.2.3 Linear RMQ

A linear space implementation is possible as described in the following: First divide v into blocks $B_1, ..., B_m$ of size $s = \frac{logn}{4}$ with $m = \left\lceil\frac{n}{s}\right\rceil$. The minimum value in each block is stored in a list. With this list, spanning block queries can be answered using the loglinear space

RMQ, which results in $O(mlogm) = O(\frac{n}{s}log\frac{n}{s}) = O(\frac{n}{logn}log\frac{n}{logn}) = O(n)$ space. For each block, the cartesian tree can be constructed in $O(n)$ time. For any two blocks with the same cartesian tree, the RMQ values are the same, so the RMQ values of all cartesian trees for all possible ranges are stored in a lookup table. Since a cartesian tree can be represented using $2s$ bits, the lookup table also requires linear space: $O(2^{2s} \cdot s \cdot s) = O(\sqrt{n}\log^2 n) = O(n)$. Therefore, for a given query $RMQ(s, e)$, depending on the positions of s and e, at most three subqueries are made: query within the block of s, query within the block of e and a spanning block query in between.

3 Implementation

There are three implementation details regarding the linear space RMQ with cartesian trees:

- Cartesian tree signatures are used to represent the cartesian trees in $2n$ bits. This representation is constructed as follows: A stack stores the rightmost path from the leaf to the root, with the leaf as the top of the stack. The list is scanned from left to right. For each element, pop the stack until the element in the top of the stack is lower than the current element (or until the stack is empty). Then push the element to the stack. In the representation, append 1 for each time an element is popped from the stack, and append 0 for each time an element is pushed to the stack. With this construction,it is not necessary to actually construct the cartesian tree, since we only need the representation as index in the lookup table.
- In the description of the data structure in the lecture, loglinear space RMQ was used to answer spanning block queries. This implementation uses instead recursive linear space RMQ for these queries until a base case of a list with 100 elements, where loglinear space RMQ is used at the end. This substantially reduces both time and space requirements for large n.

4 Experimental Evaluation

In Figure 1, it can be seen for large lists, the linear RMQ is superior to the loglinear RMQ in terms of both execution time and required space. The naive RMQ does not work because the required space is too large and does not fit the memory. The space that the linear RMQ requires is around four times smaller than the space requirement of the loglinear RMQ. Because $log_2(1000000) \approx 20$, it can be observed that the constants involved in the space complexity of linear RMQ is ≈ 5 times larger than that of loglinear RMQ. This can also be observed in Figure 2, where a list size of 4 is used. The size linear RMQ requires is also ≈ 5 times larger than loglinear RMQ, which supports the previous finding. One can also see that the loglinear RMQ requires more space than naive RMQ, but that is mainly because loglinear RMQ requires to keep the list to choose between the answers of the subqueries, whereas the naive RMQ does not need to keep the list.

In Figure 3, the space requirement and the execution time of predecessor queries with Elias Fano Coding is plotted with lists of 1000000 elements with different largest element. The time and space complexity of Elias Fano (Section 2.1 depends logarithmically on u, with which the elements in the list are bounded. The execution time scales well to u, which increased less than twice from $u_1 = 2.5e6$ to $u_2 = 1.8e19$. The space in bits also scales well, but not as good as the execution time, with an increase of 8-fold.

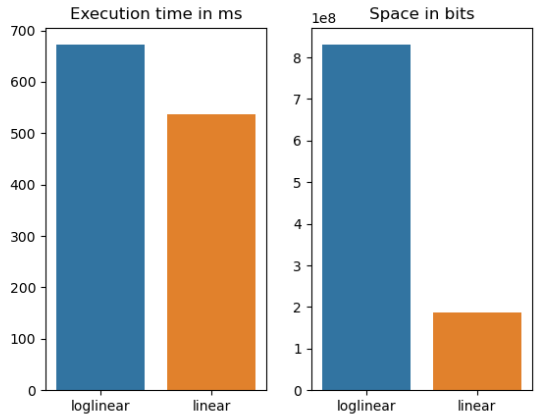

Figure 1 Execution time and required space to answer RMQ queries with a list of size 1000000

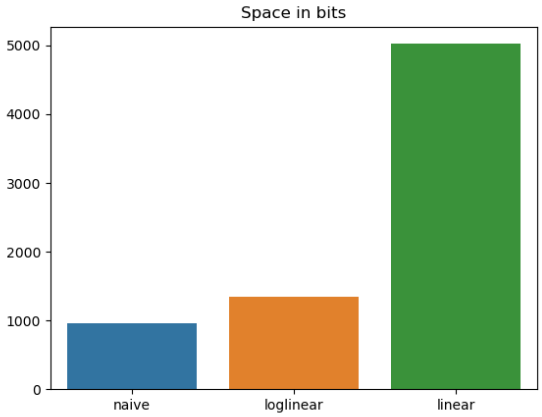

Figure 2 Required space to answer RMQ queries with a list of size 4

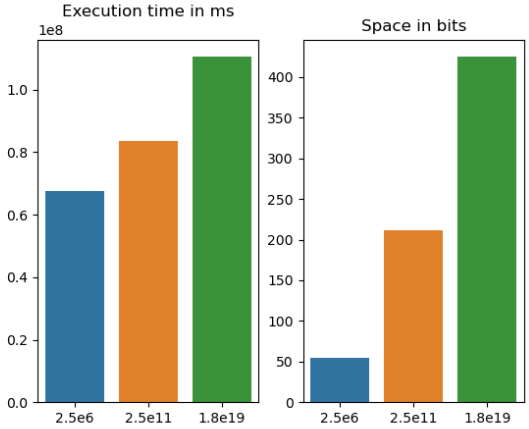

Figure 3 Execution time and required space to answer predecessor queries with a list of size 1000000 and with different largest elements.