Applied Algorithms 2019: Final Project

Martin Aumüller and Holger Dell

Published: November 20, 2019
Strict hand-in date: December 19, 2019, 13:59

This assignment has 4 pages.

Format of Hand-in This assignment can be solved in groups of up to three people. (You can choose a new group if you want.) Please submit a .zip file on learnIT, containing:

1. your code in subdirectories `part1/` and `part2/`, including all necessary non-standard libraries and classes required to compile your code, and a script (e.g., `run.sh` or `run.py`) that we can use to re-run your experiments.
2. a single `report.pdf` file containing a report of at most 4 pages written text (**11pt** font, 1in margins, a4paper, normal line spacing). The sections **must** be called "Part 1" and "Part 2" with subsections for each exercise. Plots, tables, and segments of code **do count** towards the page limit.
3. a file `authors.txt` containing a list of the group members and email addresses in the following format (and nothing else):

```
Martin Aumüller <maau@itu.dk>
Holger Dell <hold@itu.dk>
```

Oral exam. There will be an oral group exam on January 13–15, 2020. Please be at the exam room at least 15 minutes before your scheduled time. You will be examined by Holger and an external censor.

Each student will be assessed for 20 minutes. At the beginning of the exam, the group presents their project together for roughly 5 minutes. We will have your report in front of us on paper as well as your code on our computer. You can use a whiteboard or paper, or you can show your code in a terminal on our computer. However, you cannot use any kind of slides. After the 5-minute presentation, we will assess each student individually, for 10-15 minutes per student, while their fellow group members (if any) wait outside. We will ask questions about your project, and also about topics from the entire course syllabus (insofar as they are related to the intended learning outcomes). In the remaining time, the examiners discuss the grade, tell you the grade and, if you want, explain it.

Grading. The grade will be based on the project as well as the oral exam. In the project, we care about the quality of 1) your implementations, their correctness and efficiency, 2) your experiments, their design, execution and reproducibility, and 3) your evaluations, analysis, and reflections in the report.

1 Part 1: Skewed Binary Search Trees

In this exercise, you should implement a binary search data structure in three different versions, and experimentally compare them. Upon its construction, the data structure is given a set S

containing n distinct 32-bit integers. It should then support queries of the form

$$\texttt{Pred}(x) = \max\{y \in S \mid y \leq x\} .$$

That is, the data structure needs to have a method `Pred` that, on input x, returns the largest integer y with $y \in S$ and $y \leq x$.

Exercise 1. Implement three different versions of skewed binary search trees with a "balance parameter" α between 0 and 1.

1. `SortedArray`: The elements of S are stored in single, sorted array, and queries are solved using a skewed binary search. (For example in the first step, instead of jumping to position $|S|/2$ (up to rounding), you should jump to position $\alpha \cdot |S|$ (up to rounding).)
2. `SearchTree`: Each node of the tree is represented as an object that holds an element of S and has up to two child nodes. The left subtree should have an α fraction of the nodes, and the right subtree should have the remaining $1 - \alpha$ fraction.
3. `OtherArray`: The elements of S are stored in a single `int` array of length $3n$ that represents a skewed binary search tree. Each of the n nodes of the tree is stored in the array as three consecutive entries (left, key, right), where left and right are the array indices for where the left and right child nodes are stored in the array (or -1 if they don't exist), and key is the element of S represented at this node. Look at the figures in the Brodal talk (pages 21 and 26, see LearnIT), or Figures 4 and 6 in the paper by Brodal and Moruz (see LearnIT). In your report, use 2-3 sentences to explain the figures and which of the listed data structures appears to work best in their experiment. Next, based on the figure, pick one of the better-performing memory layouts for skewed binary search trees and implement it (to be clear, do not pick "inorder" as this is just a sorted array). To obtain a 12, it is necessary to implement a blocked layout of your choice (pages 23–25 in Brodal's slides) – for a 10, a non-blocked layout (page 20 in Brodal's slides) is sufficient. In the report, briefly describe the implementation details of the memory layout you have chosen.

We provide an automatic test for your implementations on CodeJudge. Each of the three programs should read the following format on standard input:

```
4
93 1094 11 -13
93 0 9999 -99 18
```

The first line contains the size n of the set S and the second line contains the set S in some unknown order. The third line contains queries x to `Pred` – let us call the number of queries q. To be clear, each query must be answered using an individual call to the function `Pred` in the respective data structure. In the end, the programs should print the following on standard output:

```
93 -13 1094 None 11
```

Exercise 2. Design, run, and evaluate an experiment comparing your three implementations with respect to the average running time per query.

In particular, remember to describe your experimental design using the proper terminology, describe and explain your a-priori hypothesis for what the experiment will show, describe and evaluate the results of your experiments, use plots with axis labels and units to visualize the obtained data, and explain in how far your hypothesis was supported by the findings. Remember to describe how your input is generated, and to use a large n and even larger q. Include a plot that shows for fixed large n and q the relationship between the balance parameter α and the average query times for `SortedArray`, `SearchTree`, and `OtherArray`.

2 Part 2: Rank-select data structures

A rank-select data structure provides access to a (static) vector of length n, containing only 0-1 values. You can assume that n is at least 64 and a power of two. The data structure supports the following operations: `Rank`(i) for $1 \leq i \leq n$ returns the number of 1s up to and including position i in the vector, and `Select`(r) returns the position of the rth 1 in the vector. Such data structures are fundamental to more complex data structures, and it is possible to implement them in a space efficient way, so that only $O(n)$ bits are used and the queries can be answered in constant time.

Exercise 3. Implement the following three variants of rank-select data structures.

- `RankSelectNaive`: the naïve data structure that uses an `int[n]` array of 0/1 values to store the data and where queries may take up to $O(n)$ time.
- `RankSelectLookup`: a smarter data structure, where you in $O(n)$ time precompute the rank of all positions and store it in an `int[n]` array. `Select` queries should be answered in $O(\log n)$ time by using a binary search on this rank array.
- `RankSelectSpaceEfficient`: a more space-efficient data structure with a parameter k, described in Section 1.3 of the paper by González et al. (see LearnIT). It uses one array R_s, where for all $i \in \{0, \ldots, \lfloor \frac{n}{32k} \rfloor\}$, the entry $R_s[i]$ is set to `Rank`$(32ki)$ in a precomputation step. `Select` queries are again implemented using a binary search on the rank.

In your report, briefly describe the approach taken for each implementation.

Your three programs **must** adhere to the following input/output format. The input is given on stdin:

```
2948679682100370091 -6730236453359443949
Rank 3
Rank 2
Rank 8
Select 1
Select 3
Rank 21
Rank 98
Select 20
```

It consist of an n-bit vector and then some rank/select queries, one per line.

More precisely, the input vector is divided into $n/64$ parts of 64-bits; each part is stored as a signed 64-bit `long`. For example, the first line above encodes a 128-bit vector (here broken into 2 lines for readability; the bits give a single total vector):

```
00101000111010111101000010001100010100000000000010001101010101011
1010001010011001011001011001101010011111001000000010000000010011
```

After the first line, each line of the input is either a rank or a select query. For each one, the program has to print one line with the corresponding correct answer.

```
1
0
2
3
9
..
```

Note that the first entry of the vector has index 1 and the last one has index n.

Exercise 4. Test the correctness of your implementations. In your report, describe how many cases you tested the implementations on, what kind of tests you created manually and what input generator you used, how you determined the correct output, and whether you ran your tests manually or in an automatic fashion (you should do the latter).

Exercise 5. Design, run, and evaluate an experiment, separately comparing the average running time of rank queries and select queries across the different implementations and for at least three choices of the parameter k. Note that n needs to be fairly large for the experiments to be meaningful (for example, much larger than the processor caches), and the number of rank and select queries needs to be reasonably large as well to get good estimates for the average running times.

For large n, report the average query times in a plot for each combination of query type, implementation, and parameter that you pick. Additionally for each implementation, determine on your machine, how large n can be until the implementation breaks.