Exam Project – Part 1

DM803 Advanced Data Structures

Dennis Andersen – deand17

Department of Mathematics and Computer Science
University of Southern Denmark

March 17, 2022

1 Introduction

The goal of the first part of the exam project was to implement the skip list and scapegoat tree data structures according to the description in their respective papers, investigate the properties of each separately, and compare them. We start by briefly discussing design choices for the implementation and then proceed to discuss experimental investigations.

2 Implementation

The skip list and scapegoat tree data structures have both been implemented using C++ and support operations to insert, search, and delete an integer key via the methods `insert`, `search`, and `remove` respectively. Duplicate keys are *not* allowed. Each method returns a `std::pair<int, bool>` with the integer being the number of key comparisons used for that call and the boolean indicating whether the operation was successful or not. In addition a `size()` function is provided, which returns the number of keys currently in the data structure.

The skip list implementation is essentially a one-to-one implementation of the pseudo code described in the paper, with the main difference being that we use a sentinel node to function as both the header, as well as the target of any NIL pointers, removing the need for `nullptr` checks. The list of forward pointers for each skip list node is implemented using a "fixed" sized `std::vector` of size *level cap*. That is, the vector is only changed by element-wise access; nothing is ever pushed or popped from the vector and thus its size never changes. This allows values for both p and *level cap* to be optionally given on the command line, with allowed ranges $(0, 1)$ and $[1, 64]$ respectively. Thus executing the following command

```
> ./skip_list 0.5 16 < example_input
```

will interpret 0.5 as the argument for p, 16 as the argument for *level cap*, and `example_input` as the file to be read from `stdin`. Values for both p and *level cap*, one or the other, or none can be given; order is irrelevant and any additional arguments are ignored. If no values are given, default values $p = 1/\mathrm{e} = 0.36788$ and *level cap* $= 32$ are used.

The scapegoat tree implementation is also essentially a one-to-one implementation of the pseudo code described in the paper, with the base binary tree implementation being based on CLRS chapter 12. The main differences with respect to the CLRS pseudo code are 1) we do not keep a parent pointer in each node, and 2) because of that we use a recursive insert method to be able to backtrack up the tree to find an α-height-unbalanced node. The base CLRS insert and remove methods are

then augmented with the balance check and rebuild methods as described in the paper, where we use equation (4.6) $i > h_\alpha(size(x_i))$ as the criterion for finding the scapegoat node. A value for α in the range $(0.5, 1)$ can optionally be given on the command line, with the default value being $\alpha = 0.55$. As with the skip list, any additional arguments are ignored.

A makefile is included and the default target builds the programs for both the skip list and the scapegoat tree as `skip_list` and `scapegoat_tree` respectively. In addition, running `make test` will build both programs and run each in turn using the example input from the project description as a quick test that everything compiled and works as expected. The implementation has been tested to work on the computer lab machine `imada-106333`. Also included is a small shell script `test.sh`, taking the following options:

- `-h` Print this Help.
- `-p` Value of p for Skip List. Defaults to 0.36788.
- `-a` Value of α for Scapegoat Tree. Defaults to 0.55.
- `-n` Number of keys to insert. Defaults to 1024.
- `-k` Number of times to run search for n keys. Defaults to 1.

This script generates[1] k test files, and for each file, two uniformly random samples of the integers $0, \ldots, n-1$ are generated, the first of which is used for n insert operations followed by n search operations using the second sample. The script then runs the `skip_list` and `scapegoat_tree` programs on each of the k input files, followed by a post processing[2] step, which outputs the results of the test to `stdout`.

3 Experimental Results

We now want to investigate and understand the skip list and scapegoat tree data structures. We first examine whether the average search complexity of our implementation matches the expected average search complexity. We then look at the variation in search complexity, and finally for scapegoat trees specifically, we examine the relationship between the frequency and size at which trees are rebuilt. We measure complexity as the number of times two keys are compared. We have used the script `test.sh` described in the previous section to run our tests. Also, note that the delete operation has not been included in our analysis, but has been tested for correctness using the example input from the project description.

3.1 Average Search Complexity

For testing the average search complexity, we have used $k = 3$ for all our tests, meaning for each input size $n \in \{2^{10}, 2^{11}, \ldots, 2^{15}\}$, we have made 3 test runs and then calculated the average number of key comparisons across the 3 runs. We tested the skip list using values of $p \in \{1/2, 1/\mathrm{e}, 1/4, 1/8, 1/16\}$ and the scapegoat tree using values of $\alpha \in \{0.55, 0.6, 0.65, 0.7, 0.75\}$.

Figure 3.1 shows the complexity as a function of n for the search operation, with the left plot showing the results for the skip list and the right plot the results for the scapegoat tree. Both plots use a lg-scale on the x-axis.

The expected average complexity for the search operation for both data structures is $\lg n$, thus we expect our data points to form a line, which is indeed the case for the scapegoat tree and mostly for the skip list as well. For the skip list however, with $p = 1/16$, we start to observe a more linear

[1] Using `generate_input_files.py`
[2] Using `postprocess.py`

behaviour. This is of course because each node will have a higher probability of containing several levels, meaning fewer nodes are likely to be skipped while searching, and so we get behaviour

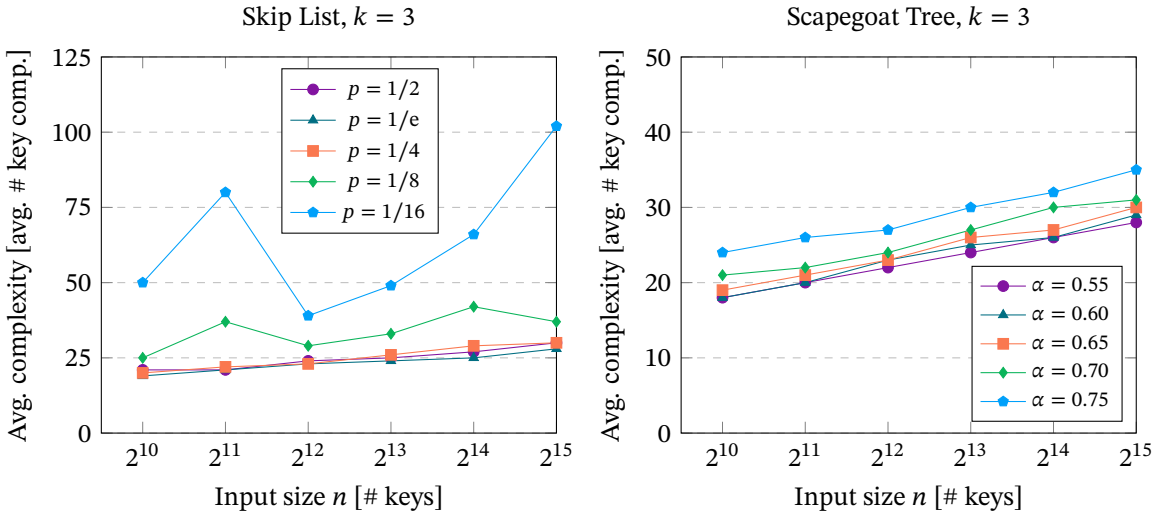

Figure 3.1 Average search complexity (number of key comparisons) for our implementation of the search operation for the skip list and scapegoat tree. Both plots use a lg-scale x-axis and show the average search complexity across $k = 3$ test runs for increasing input sizes n. Left: Results for skip list using decreasing values for p. Shows logarithmic behaviour for search for all but $p = 1/16$. Right: Results for scapegoat tree using increasing values for α and shows logarithmic behaviour for all values of α.

that more resembles that of an ordinary linked list.

Using logarithmic regression[3] in R, we can estimate the constant term in front of the logarithm for both data structures for respective values of p and α, with the results shown in table 3.1.

Table 3.1 Logarithm Constant Terms

	Skip list			Scapegoat tree	
p	c	p-value	α	c	p-value
1/2	1.8286	0.0008	0.55	2.0000	0.0000
1/e	1.6570	0.0003	0.60	2.1429	0.0001
1/4	2.1143	0.0002	0.65	2.1714	0.0000
1/8	2.2570	0.1315	0.70	2.2000	0.0002
1/16	6.5140	0.2906	0.75	2.1714	0.0001

From table 3.1, we observe that both the smallest and largest constant terms come from using the skip list. With a p-value of 0.0003, the smallest term of 1.657 is obtained with high confidence when using $p = 1/e$, and the largest term of 6.514 is obtained when using $p = 1/16$, although with a lower confidence (p-value of 0.2906). Thus, we again observe the probabilistic nature of the skip list in that the constant terms vary quite significantly when compared to the scapegoat tree, where the constant terms are more stable and consistently lie in the range 2 to 2.2.

We conclude this section by noting that values $p = 1/e$ and $\alpha = 0.55$ for skip list and scapegoat tree respectively, seem to show the best performance for each respective data structure, which is the reason these values are used as the default values in our implementation.

[3] File `log_reg.R`

3.2 Variation in Search Complexity

We now look at the variation in search complexity for the skip list and scapegoat tree. That is, for n searches, we make a histogram counting the number of searches s_j that used i key comparisons and then accumulate the percentage of searches that had complexity i as shown in the plots in figure 3.2. For this test we used $n = 2^{10}$ (figure 3.2 top) and $n = 2^{15}$ (figure 3.2 bottom) and we again made

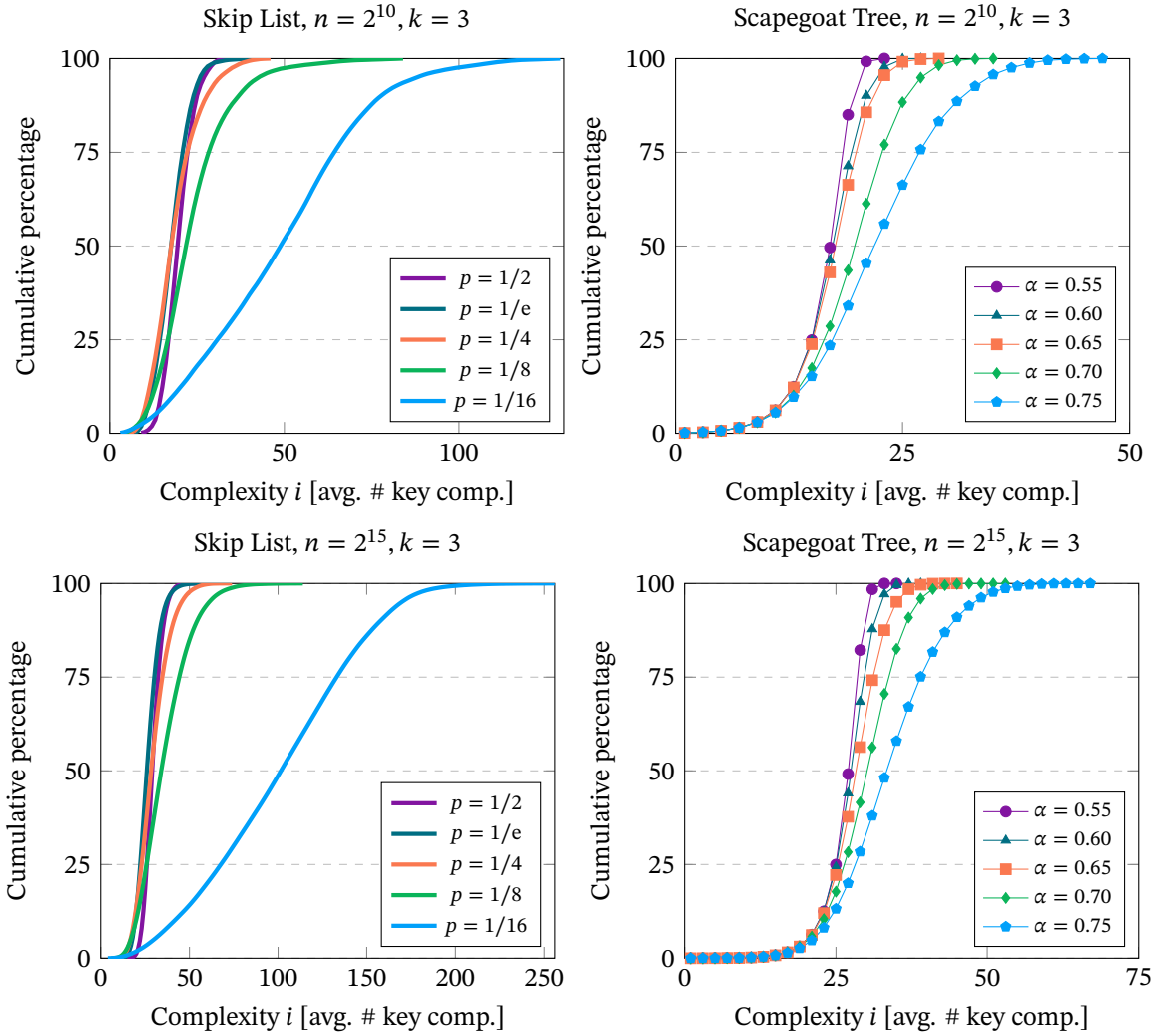

Figure 3.2 Variation in average search complexity (number of key comparisons) for our implementation of the search operation for the skip list and scapegoat tree. Each plot shows the variation in average search complexity across $k = 3$ test runs as the cumulative percentage of searches that used i comparisons or less. Top left: Results for skip list using $n = 2^{10}$ and decreasing values for p. A fair amount of variation is observed for all values of p except $p = 1/16$ for which the variation is much greater. Top right: Results for scapegoat tree using $n = 2^{10}$ and increasing values for α. A relatively small amount of variation is observed for smaller values of α. Bottom left: Results for skip list using $n = 2^{15}$ and decreasing values for p. An amount of variation comparable to $n = 2^{10}$ is observed for all values of p except $p = 1/16$, for which the variation is approximately doubled. Bottom right: Results for scapegoat tree using $n = 2^{15}$ and increasing values for α. An amount of variation comparable to $n = 2^{10}$ is observed, but as before, the variation increases for larger values of α.

$k = 3$ test runs and used the average number of comparisons of the 3 runs. In addition, we used the same values for p and α as previously.

The variation in average search complexity is shown in figure 3.2, which contains four plots. Figure 3.2 left shows the results for skip list using $n = 2^{10}$ (top) $n = 2^{15}$ (bottom) and and decreasing

values for p. For $n = 2^{10}$ we observe a fair amount of variation concentrated in the range 10-50 for values of p between 1/2 and 1/8. For $p = 1/16$ however, the variation is much larger, extending all the way to 133, which was the maximum number of comparisons recorded for this test. For $n = 2^{15}$, the variation for values of p between 1/2 and 1/8 are comparable to $n = 2^{15}$ and as before concentrated in the 10–50 range. For $p = 1/16$, the variation is even larger though, extending to 256 comparisons. Figure 3.2 right shows the results for scapegoat tree using $n = 2^{10}$ (top) $n = 2^{15}$ (bottom) and and increasing values for α. For both $n = 2^{10}$ and $n = 2^{15}$ we observe a relatively small

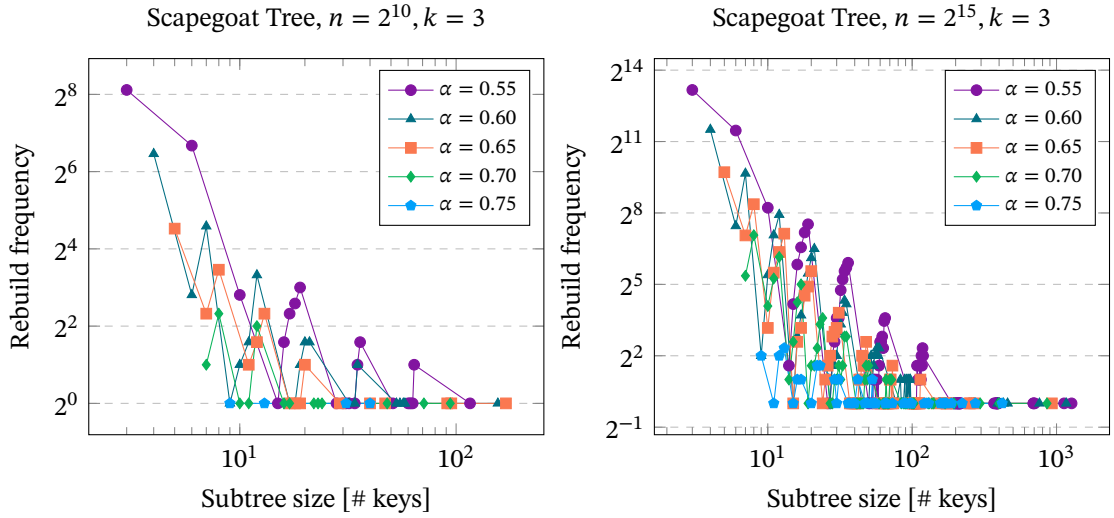

Figure 3.3 Frequency and size relationship of scapegoat tree rebuilds with increasing values of α. Both plots uses a log-lg-scale. Left and right show the frequency and size relationship of rebuilds with $n = 2^{10}$ and $n = 2^{15}$ respectively. We generally observe that smaller values of α trigger frequent rebuilds with small subtrees and that rebuilds involving a large number of nodes rarely occur.

and comparable amount of variation for all values of α, and comparing the two data structures, the scapegoat tree in general shows less variation in the average search complexity when compared with the skip list.

3.3 Frequency and Size Relationship of Scapegoat Tree Rebuilds

Our final point of investigation is on the frequency and size relationship of scapegoat tree rebuilds. As with the previous tests, we have used the average of $k = 3$ test runs. To investigate the frequency and size of rebuilds, we temporarily modified the program to output the size of the subtree whenever a rebuild occurred and then used Python[4] to make a histogram over the frequency of rebuilds of each size, with the results shown in figure 3.3.

Figure 3.3 left and right show the frequency and size relationship of rebuilds with $n = 2^{10}$ and $n = 2^{15}$ respectively. As is to be expected, we generally observe that smaller values of α trigger a large number of rebuilds with small subtrees of size less than 10 and that large rebuilds involving more than 100 nodes occur quite infrequently.

As a final note, although not reflected in any of the tests shown here, we did notice a significantly larger runtime of the scapegoat tree program compared to the skip list program, and we mainly believe this is due to the heavy reliance of recursion. Thus we would not be surprised to find that the statement from the scapegoat tree paper claiming that runtimes were reduced by 20–30% to be true.

[4]File `freq.py`