NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Scapegoat trees:
a comparative performance assessment

Vasilios I. Venieris

Supervisor: **Kostas Chatzikokolakis,** Assistant Professor

ATHENS

MAY 2022

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Δέντρα scapegoat: μια συγκριτική αξιολόγηση επιδόσεων

Βασίλειος Ι. Βενιέρης

Επιβλέπων: **Κώστας Χατζηκοκολάκης**, Αναπληρωτής Καθηγητής

ΑΘΗΝΑ

ΜΑΪΟΣ 2022

BSc THESIS

Scapegoat trees: a comparative performance assessment

Vasilios I. Venieris
S.N.: 1115200800225

SUPERVISOR: **Kostas Chatzikokolakis,** Assistant Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Δέντρα scapegoat: μια συγκριτική αξιολόγηση επιδόσεων

Βασίλειος Ι. Βενιέρης
Α.Μ.: 1115200800225

ΕΠΙΒΛΕΠΩΝ: **Κώστας Χατζηκοκολάκης**, Αναπληρωτής Καθηγητής

ABSTRACT

This Thesis presents and analyzes an alternative tree scheme for the efficient balancing of binary search trees. Scapegoat trees are loosely balanced and restructure parts of themselves on certain conditions. The amortized time complexity for each INSERT or REMOVE operation is $O(logn)$, while the worst-case real-time complexity of a FIND one is $O(logn)$. Scapegoat trees, unlike most self-balancing BST implementations, do not require extra data (e.g. colors, weights, heights) in the tree nodes. Various metrics and tests are used to compare AVL and scapegoat trees on their functionality. Finally, we provide data as to the performance of scapegoat trees in potential real-time applications.

SUBJECT AREA: Data structures

KEYWORDS: Scapegoat, binary search tree, self-balancing, AVL, amortized time

ΠΕΡΙΛΗΨΗ

Η εργασία αυτή παρουσιάζει και αναλύει μια εναλλακτική μέθοδο για την αποδοτική εξισορρόπηση δυαδικών δέντρων αναζήτησης. Τα δέντρα scapegoat είναι ελαφρώς ισορροπημένα και ανακατασκευάζουν μέρη τους υπό ορισμένες συνθήκες. Η amortized χρονική πολυπλοκότητα για κάθε λειτουργία INSERT ή REMOVE είναι $O(logn)$, ενώ η worst-case πραγματική χρονική πολυπλοκότητα μιας FIND είναι $O(logn)$. Τα δέντρα scapegoat, σε αντίθεση με τις περισσότερες υλοποιήσεις αυτεξισορροπούμενων BST, δεν απαιτούν επιπλέον δεδομένα (π.χ. χρώματα, βάρη, ύψη) στους κόμβους των δέντρων. Χρησιμοποιούνται διάφορες μετρικές και δοκιμές για τη σύγκριση των δέντρων AVL και scapegoat ως προς τη λειτουργικότητά τους. Τέλος, παρέχουμε δεδομένα ως προς την επίδοση των δέντρων scapegoat σε πιθανές εφαρμογές πραγματικού χρόνου.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Δομές δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Scapegoat, δυαδικό δέντρο αναζήτησης, αυτεξισορρόπηση, AVL, amortized χρόνος

ACKNOWLEDGMENTS

I would like to thank my thesis supervisor, Kostas Chatzikokolakis, whose suggestions and instructions have significantly contributed to the completion of this Thesis.

CONTENTS

PREFACE 13

1. INTRODUCTION 14

1.1 Binary search trees: characteristics and self-balancing schemes 14

1.2 Scapegoat trees: the theory behind them 14

1.2.1 How they balance themselves 14

1.2.2 How to implement them 15

1.3 Implementing AVL and SG trees for testing purposes 15

2. SINGLE-OPERATION PERFORMANCE ASSESSMENT 16

2.1 Setting up the test 16

2.1.1 AVL depth and rotations 16

2.1.2 SG depth vs AVL depth for various insertion configurations 16

2.2 Basic BST operations 17

2.2.1 INSERT(x): performance analysis 17

2.2.2 REMOVE(x): performance analysis 18

2.2.3 REMOVE(x): testing optimal upper bounds for q 20

2.2.4 FIND(x): performance analysis 21

2.3 Analysis of correctness for INSERT(x) and REMOVE(x) 22

2.3.1 REMOVE(x): amortized logarithmic cost 22

2.3.2 INSERT(x): amortized logarithmic cost 22

3. MULTI-OPERATION PERFORMANCE ASSESSMENT 24

3.1 Workload diversification: defining rules for the test 24

3.2 Scaling and bias parameters 24

3.3 Testing for small datasets and sessions 24

3.4 Computational hurdles in bigger datasets and how to avoid them 25

3.5 Results and performance assessment 26

4. DISCUSSION AND FUTURE WORK 27

5. CONCLUSIONS 28

TABLE OF TERMINOLOGY 30

ABBREVIATIONS - ACRONYMS 31

REFERENCES 32

LIST OF FIGURES

Figure 1: AVL depth and rotations 16

Figure 2: AVL/SG average depth 17

Figure 3: Insertion performance 18

Figure 4: Removal performance (Same-order removal) 19

Figure 5: Removal performance (Shuffled removal) 19

Figure 6: Removal performance (Unordered insertion, same-order removal, breakdown by factor) 20

Figure 7: Search performance (Same-order search) 21

Figure 8: Search performance (Shuffled search) 22

Figure 9: Insertion restructuring for $ALPHA$ = 0.6667 (Unordered insertion, breakdown by set size) 23

Figure 10: Session performance for 100K integers (breakdown by query ratio and implementation) 25

Figure 11: Session composition for X5 query ratio (breakdown by set size and query type) 25

Figure 12: Asymptotic session performance for 100 integers, X20K query ratio (breakdown by implementation) 26

Figure 13: Asymptotic session composition for 100 integers, X20K query ratio (breakdown by query type) 26

LIST OF TABLES

Table 1: Table of terminology 30

Table 2: Abbreviations - Acronyms 31

PREFACE

When I first encountered the concept of scapegoat trees, I was nothing short of intrigued. However, delving into detailed performance evaluation proved to be extremely difficult; most sources I came across completely lacked any comparative analysis, while others, like the early works of Galperin-Rivest and Andersson, were limited, owing to their contemporary computational means. This Thesis is an attempt to fill that void.

1. INTRODUCTION

1.1 Binary search trees: characteristics and self-balancing schemes

Binary search trees are used to store items (values, key-value pairs, strings etc.) in sorted order, and possess the ability for fast insertion, deletion and look-up, similar to sorted arrays, however, without the static memory limitations that the use of such linear data structures incur. The crucial problem that BSTs face is that, given a monotonous set of items, BSTs tend to degrade themselves to linked lists and, thus, losing their edge over their linear counterparts; therefore, ever since their inception in the 1960s, various tree-balancing schemes and techniques have emerged over the years to counter said tendency [1]. The most prevalent implementations of self-balancing BSTs are AVL trees and Red-Black trees, each with their own pros and cons:

- AVL trees excel at minimizing look-up times, essentially creating near-perfect BSTs, that is, binary tree structures with each internal node having 2 children and external nodes having none, at the cost of additional operations per insertion and deletion and higher memory cost per node,
- Red-Black trees excel at minimizing insertion and deletion times, at the cost of somewhat more loosely-constructed BSTs, making them ideal for applications that favor insertions and deletions, such as Linux kernels.

This Thesis will attempt to examine a less popular, yet interesting BST implementation: the scapegoat tree.

1.2 Scapegoat trees: the theory behind them

The name itself is based on the ancient practice of “singling out a person or group for unmerited blame and consequent negative treatment”, that is, a scapegoat. When blame is established, the scapegoat is left to fix the problem. Scapegoat trees belong to a wider range of BSTs, called α-height-balanced trees [2] [3] [4], whose nodes are defined by the following relaxed height balance criterion:

$$size(node.child) \leq a \cdot size(node)$$

The above inequality provides 2 interesting corollaries:

- None of the 2 childrens’ subtrees can own more than α·100% of the nodes contained within the parent’s tree; it sets a hard cap on both childrens’ tree sizes.
- The height balance factor (α, or $ALPHA$) is bounded between (0.5, 1).

1.2.1 How they balance themselves

Scapegoat trees balance themselves through partial rebuilding operations, during which, an entire subtree is deconstructed and then rebuilt into a perfectly balanced one. One could argue that rebuilding a subtree would be quite the suboptimal balancing method, since that would occasionally require that the whole tree be rebalanced; thus, the operation would essentially take $\Theta(n)$ worst-case time, making it considerably worse than AVL trees, which would take $\Theta(1)$ worst-case for their rebalancing operations respectively. However, as we will show in Chapter 2, where insertions and removal will be further analyzed as to their time complexity, this new rebalancing operation takes essentially $O(1)$ worst-case real-time, establishing it as a very competitive alternative to most BST implementation schemes.

1.2.2 How to implement them

Implementation-wise, they are quite simple: in addition to storing n, the size, that is, the number of nodes in the tree, a separate counter q is also kept, that maintains an upper-bound on the number of nodes and they must both obey the following inequality:

$$n \leq q \leq 2n \text{ [5].}$$

- q indicates a loose estimation on the tree's maximum permissible height, which is calculated as follows: $h_{max} = log_{1/a}q$ [4]. The logarithm's base occurs from the fact that every node is supposed to have at most α·100% of its parent's size.
- When the SG tree is initialized, $q = n$. Every time a successfull insertion occurs, q is increased by one, therefore increasing maximum permissible height. If the newly inserted item's path depth exceeds h_{max}, it causes the tree to increase in height too much, effectively violating the height constraint and, by extension, our α-criterion. The unbalanced subtree rooted somewhere along that path is then found and rebuilt.
- When an item is successfully removed, we check whether q is still bounded from above. If q surpasses $2n$ (due to the tree having lost nodes), the entire tree is rebuilt in $\Theta(n)$ worst-case real time. Then q is set to n and, consequently, the tree's maximum height drops, since the SG tree now resembles a near-perfect BST.
- Since both n and q are increased by one during insertions, if no removal is done at all, the initial equality always holds. Therefore, if our BST needs to support only insertions and searches as basic operations, we could simplify our implementation even further, by omitting q and its upper bound. Finally, $h_{max} = log_{1/a}n$.

Furthermore, scapegoat trees have no additional per-node memory overhead, unlike AVL trees, whose nodes store their balance factor in the form of integers.

1.3 Implementing AVL and SG trees for testing purposes

One can't help but wonder whether scapegoat trees have anything new to offer to the table, since AVL trees are well-established and have been thoroughly studied ever since their advent in the 1960s. Therefore, in order to measure the performance of SG trees, one such implementation was extensively compared to an AVL one. To make our comparative assessment as unbiased as possible, both implementations were developed in C language and we have utilized identical helper functions for both trees, so as to keep any code dissimilarities as few as possible and reduce the chances of one outperforming the other due to code discrepancies.

2. SINGLE-OPERATION PERFORMANCE ASSESSMENT

2.1 Setting up the test

We chose to put our SG tree implementation against a typical AVL and compare the structures they produce, as well as their performance. To do that, we have 3 different metrics that will be combined and yield us the results we need:

- basic BST operations: INSERT(x), REMOVE(x), FIND(x),
- two kinds of initial datasets: ordered and unordered, essentially sorted and randomly (uniformly) spread,
- two kinds of approaches upon the dataset: same-order and shuffled; that is, we remove/find items either in the same order we inserted them based on the initial dataset or at random.

2.1.1 AVL depth and rotations

At first, we computed the average depth of AVL and SG trees after inserting 1M integers, both monotonically and at random. As shown in Figure 2, the AVL tree achieves a far lower depth than the SG one, both with ordered and unordered sets of integers, with depths 17.95 and 18.33 respectively. It should also be noted that set monotonicity is important when it comes to rotations: ordered items require on average more rotations per inserted item compared to unordered ones. However, those extra 0.3 rotations per item contribute to the tree becoming near-perfect, achieving identical average depth to a tree created by bulk initialization, shown in Figure 1.

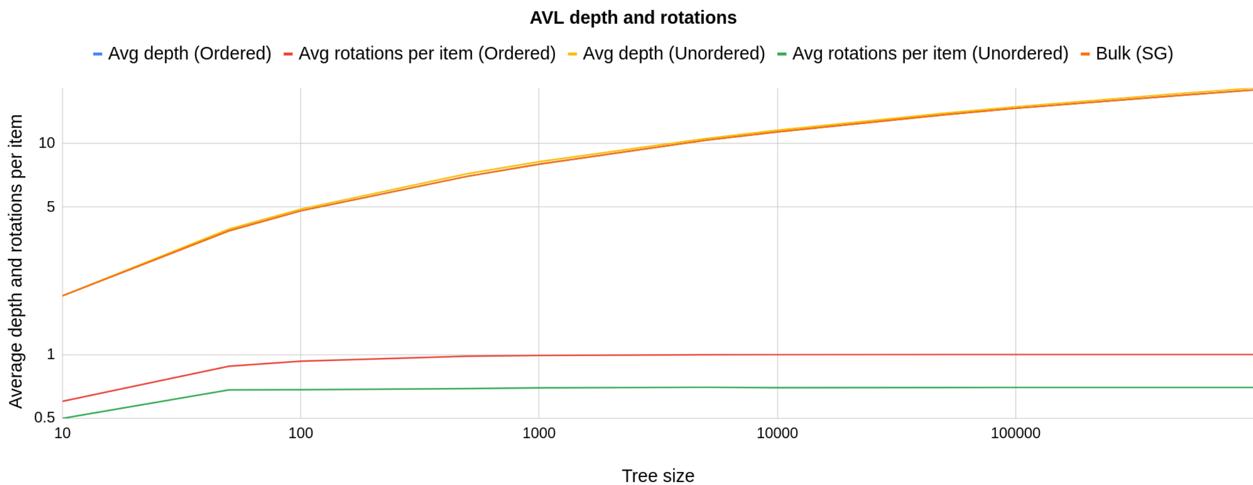

Figure 1. Average depth of an item and average number of rotations per insertion in an AVL implementation. AVL performs on average 43% additional rotations per out-of-order item and it results in the tree achieving near-optimal search times.

2.1.2 SG depth vs AVL depth for various insertion configurations

On the other hand, the SG tree acts differently, based on the $ALPHA$ coefficient. At low $ALPHA$, the SG tree essentially behaves like a quasi-AVL, where every node's child contains, on average, half of the node's subtree [4], while at high $ALPHA$, it behaves like an unbalanced BST. Figure 2 shows exactly that: inserting the items monotonically causes the SG tree to slowly degrade to a linked list, whereas inserting them randomly causes the tree to degrade until about 0.66 alpha, after which it behaves like a randomized BST.

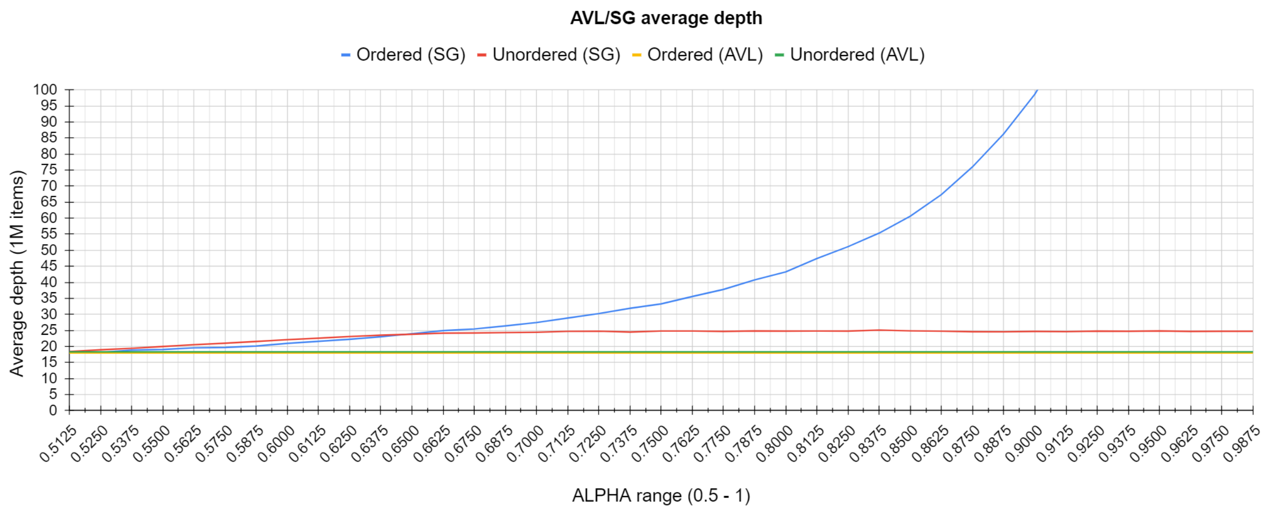

Figure 2. Implementation comparison on average depth. SG tree's is similar for ordered and unordered items, up until about 0.66, when unordered items depth stabilizes, whereas ordered item depth increases exponentially.

2.2 Basic BST operations

We tested 3 operations, common to BST implementations: INSERT(x), REMOVE(x) and FIND(x). An additional operation was implemented for the SG tree, called BULK(x[1,..,n]), which, during the tree's initialization, receives a set of ordered items and creates a perfectly balanced BST in $\Theta(n)$ real-time. The extra operation was created in order to detect and compare differences in performance between off-line and on-line insertions. So, essentially, BST initialization through bulk insertion produces the same structure as an AVL tree given ordered items one by one.

2.2.1 INSERT(x): performance analysis

Figure 3 demonstrates INSERT(x)'s different performance for each BST implementation and integer set:

- Ordered insertion for AVL is considerably better than SG tree. It achieves that by performing 1 left rotation in every insertion, as shown in Figure 1. That essentially forms a perfect tree, which in turn minimizes insertion times. Unordered insertion, on the other hand, is considerably worse, even though the AVL tree performs on average fewer rotations per integer.
- Ordered SG insertion appears to behave much worse, as $ALPHA$ moves towards the two extremes; that is due to the following reasons:
    - The lower $ALPHA$ becomes, the more frequent subtree rebalancing operations become, hence the increasing times.
    - The higher $ALPHA$ becomes, the more degraded the tree becomes, which leads to increasingly linear-like insertion times.
- Unordered SG insertion times are evidently fluctuating due to the random way items are inserted. However, for $ALPHA$ lower than 0.6, SG appears to be slightly faster at inserting, due to the lower average depth of the tree. Those two distinct performance levels should become clearer for bigger datasets, although checking it

would entail impractically high running times and/or more extensive computational capabilities.

- Bulk insertion is obviously the fastest way to insert items, since all 1M items are inserted off-line in $\Theta(n)$ real-time. This essentially acts as our experimental low bound, since no other operation or dataset order can achieve that performance; even though AVL, given ordered items, can build the same near-perfect tree, log linear time for insertions plus $\Theta(n)$ real-time for rotations incurs massive time penalties. In practice, for each of the 3 basic BST operations (INSERT(x), REMOVE(x), FIND(x)), the closer their curves approach their respective bulk-initialized operation performance, the better. Figure 4 and 7, SG ordered removal and search respectively are typical examples of said near-optimal behavior.

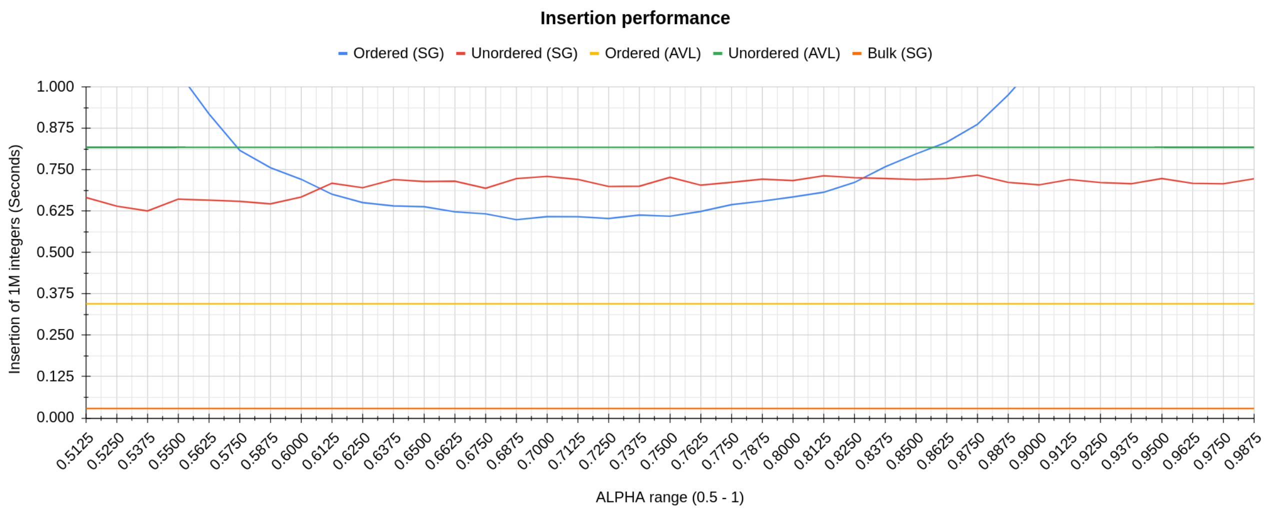

Figure 3. Implementation comparison on insertion. AVL times form a very wide band, whereas SG tree seems to be somewhat consistent, for ALPHA between 0.55 - 0.85. At the edge of the chart, ordered SG times increase exponentially.

For the next 2 standard BST operations, an extra step was added: items were removed or searched for not only in the same order they were inserted, but also at random.

2.2.2 REMOVE(x): performance analysis

The following figures show how both implementations fared when items were removed from the BSTs in the same order they were initially inserted and at random:

- When it comes to ordered removal (Figure 4), SG is superior to AVL at both ordered and unordered initial insertions. In fact, SG tree's ordered removal time appears to be optimal, as it coincides with the removal time for bulk insertion. That comes down to the way subtree rebalancing works in SG trees: during ordered insertions, when rebalancing is triggered, the scapegoat always happens to be at the root, essentially restructuring the entire tree. Therefore, when the first removal occurs, the tree already has minimal height.

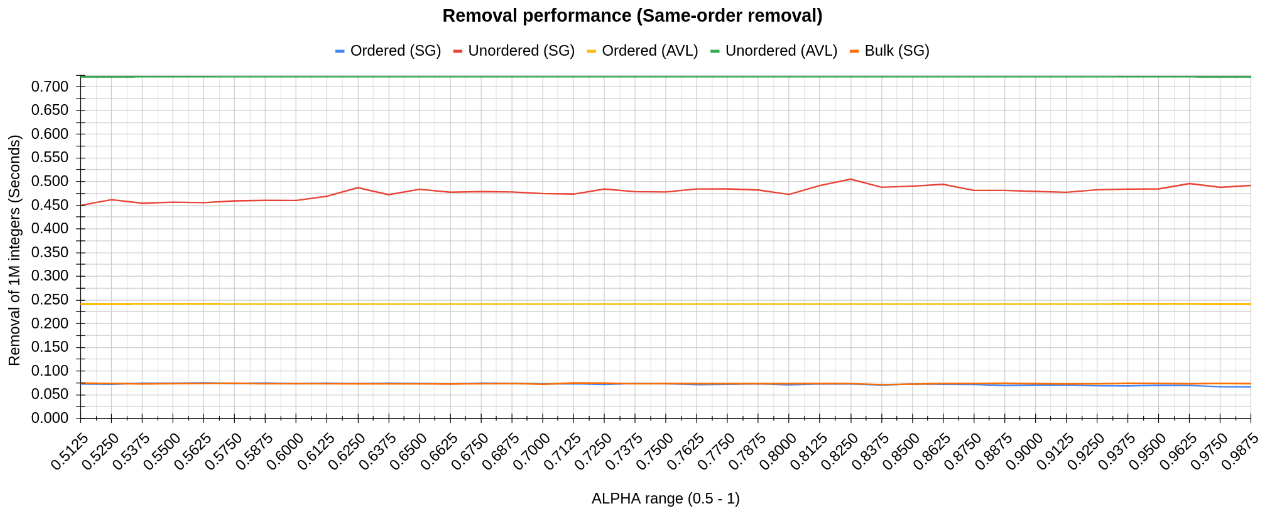

Figure 4. Implementation comparison on same-order removal. Both implementations appear to have a 450ms band gap between their respective ordered and unordered sets, at 1M items. Similar results occurred in other set sizes as well.

- In Shuffled removal performance (Figure 5), the trend of SG superiority continues, but with a few twists:
    - The time gap between AVL's ordered and unordered insertion is now much smaller.
    - SG tree's unordered insertion removal has significant time fluctuations, but one can clearly distinguish 2 separate levels forming, when $ALPHA$ becomes low enough. That can, again, be attributed to the tree's tendency of behaving like a weakly balanced AVL tree.
    - What's interesting is the tree's performance after ordered insertions: at low $ALPHA$, the tree is effectively a perfect BST. However, as $ALPHA$ increases, removal time increases exponentially in respect to $ALPHA$. That can be explained by the fact that, because of the removals' randomness, the tree is filled with inconsistent branch depths, further degrading its structure to a list.

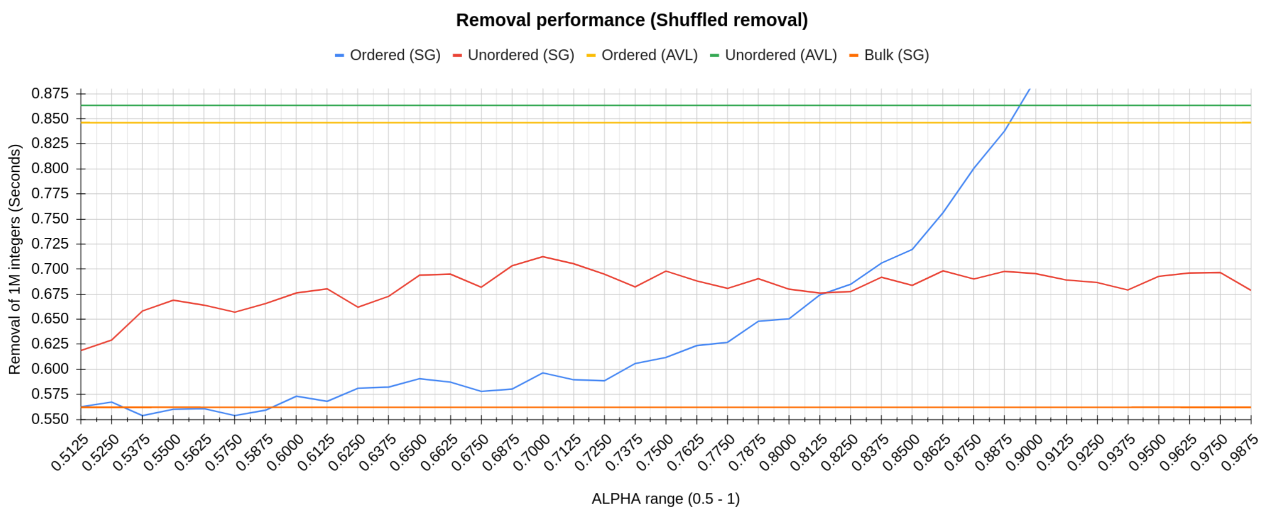

Figure 5. Implementation comparison on shuffled removal. SG tree completely dominates over AVL, even for extremely high $ALPHA$ values.

2.2.3 REMOVE(x): testing optimal upper bounds for q

Bibliography that was reviewed contained various approaches as to what the upper bound for q should be: contemporary authors, like Pat Morin, use $2n$ [5], while others, including Igal Galperin and Ronald Rivest, both prominent contributors to the study of α-height-balanced BSTs and inventors of the SG tree, use $\frac{n}{a}$ [2] [4]. Opting for the latter gives $2n$, for $ALPHA$ sufficiently close to 0.5; however, as was shown in Figure 3, that comes at a heavy insertion cost. On the other hand, Morin's choice, even though satisfying $O(logn)$ amortized time, seems somewhat arbitrary, without any substantial explanation. Thus, we decided to follow Morin's move and chose various values ourselves, in order to compare the performance differences during removal.

Figure 6 demonstrates how REMOVAL(x) operations' performance fluctuates, depending on the coefficient used in our implementation. The percentages in the legend indicate the amount of successive removals the SG tree can handle in respect to the number of initial items, before it rebalances itself from the root and resets q:

- As is clearly depicted, most coefficients seem to be forming an optimal performance band between 5 and 1.67, without any noteworthy deviation. Bigger coefficients up to 25 were also tested, but none exhibited any further improvement to performance.
- 1.2 seems to cause performance to deteriorate notably, pointing to the substantial increase in rebalancing operations, as the legend suggests.
- At 1.11, performance has worsened even further, completing all removals at x1.5 the optimal band's time.
- The band's performance is evidently affected by a sufficiently low $ALPHA$ up to 0.6. That can be attributed to the tree's initial rigid structure, which influences subsequent shapes caused by the removals.

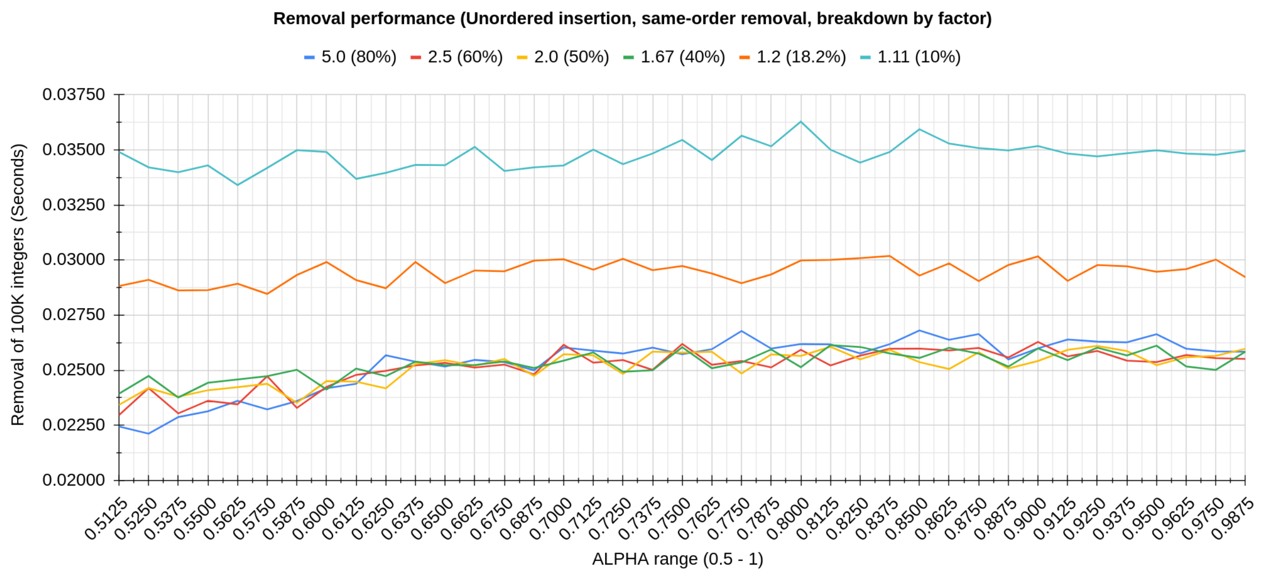

Figure 6. Scapegoat tree: upper bound for q. Although $q \leq k \cdot n,\ \forall\, k > 1$ must always be true, we can see that, after a certain k value, performance remains unchanged.

2.2.4 FIND(x): performance analysis

Unlike REMOVE(x), FIND(x) operations are significantly more consistent as to their performance for both same-order and shuffled search. Figure 7 is indicative of that consistency:

- AVL and SG trees exhibit similar performance in same-order search for low $ALPHA$. This is, of course, to be expected, because of the SG tree's rigid structure constraint.
- An interesting finding to be noted is the fact that the SG tree appears to have evidently lower average access time when bulk-initialized, compared to the AVL one; that is bizarre at first glance since, as is shown in Figure 1, AVL trees, given ordered items, produce perfect binary structures, just like bulk initialization. After initial tests, a hypothesis arose that hardware (cache, in particular) is to be blamed.
- As previously shown, increasing $ALPHA$ exponentially increases average search time for SG trees, for ordered insertions. In spite of that, at low $ALPHA$ values, the SG implementation outperforms the AVL one.
- Unordered insertion produces similar trees, even though the SG tree is more susceptible to randomness.

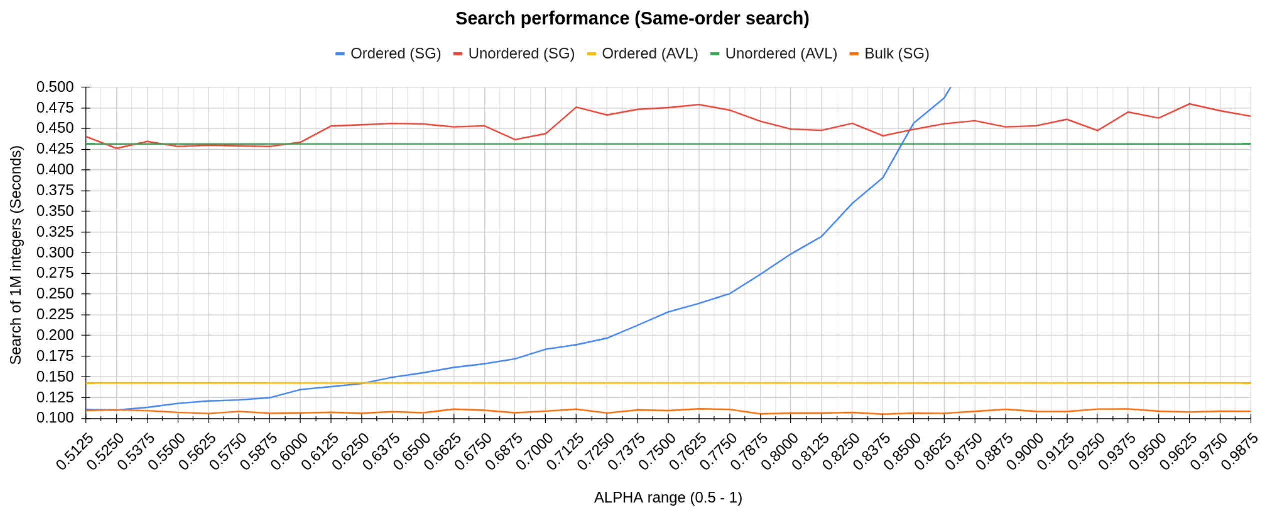

Figure 7. Implementation comparison on same-order search.

When items are accessed randomly, a similar pattern emerges, but with a few key differences, as illustrated in Figure 8:

- SG tree's average access times are considerably noisier, typical of the implementation's behavior.
- Near-identical SG tree performance to AVL at low $ALPHA$ is affirmed. In fact, there exist 3 distinct behaviors:
    - Up to about 0.6 $ALPHA$, it is virtually impossible to distinguish between SG and AVL trees,
    - A transition band between 0.6 and 0.675, during which access time severely deteriorates,
    - At 0.675, performance is somewhat stabilized, despite the frequent noise.
- AVL performance averages for ordered and unordered insertion swap places.

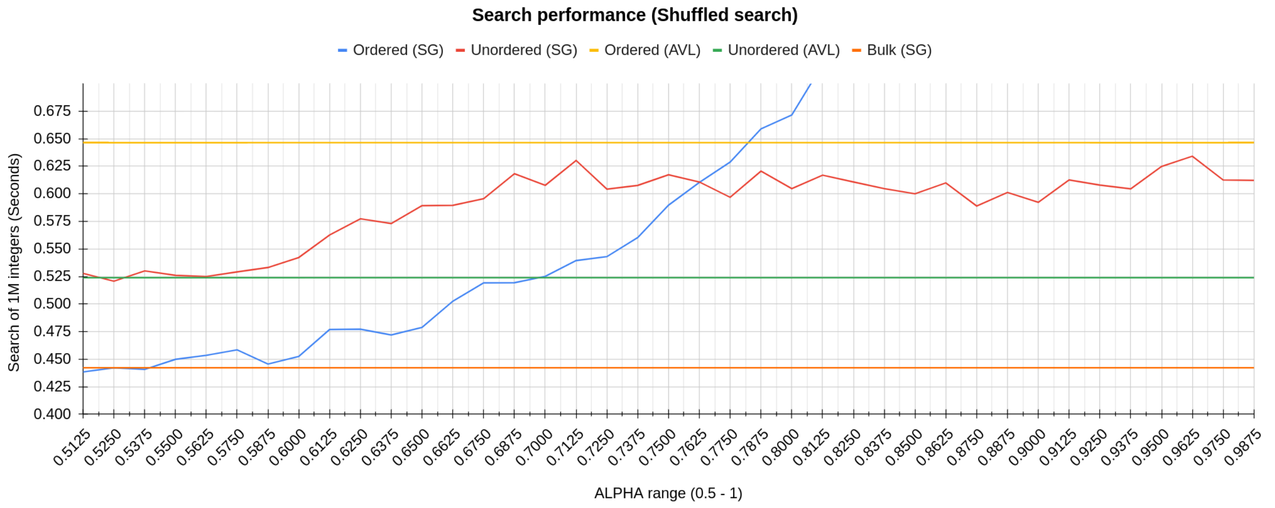

Figure 8. Implementation comparison on shuffled search. SG unordered search appears to be stabilized after 0.675, just like its average depth in Figure 1.

2.3 Analysis of correctness for INSERT(x) and REMOVE(x)

As has been previously claimed, INSERT(x) and REMOVE(x) have $O(logn)$ amortized time, which differs from AVL's $O(logn)$ worst-case real time complexity. Therefore, we are going to examine those 2 operations up close. For REMOVE(x), we chose to present the aggregate analysis approach, whereas for INSERT(x), we decided to compute and store the unbalanced subtree sizes and the frequency in which they appeared and, based on these data, infer about insertions' time complexity.

2.3.1 REMOVE(x): amortized logarithmic cost

Suppose a total rebalancing has just occurred during a removal. Consequently, q is set to n. Additionally, suppose that the upper bound for q is $k \cdot n$, with $k \in R^{+}, k > 1$. Essentially, we can perform $\frac{(k-1)\cdot n}{k} - 1$ removals before the next total rebalancing operation. Using aggregate analysis, the amortized cost of a removal, $\widehat{T}$, is:

$$\widehat{T}(n) = \frac{\sum_{1}^{c} O(logn) + \Theta(n)}{c},\ c = \frac{(k-1)\cdot n}{k} \Rightarrow \widehat{T}(n) = O(logn) + \frac{k}{k-1} \cdot \Theta(1) = O(logn)$$

We have thus proven that, no matter the coefficient k, removals will always bear logarithmic cost, in an amortized sense, which is also hinted at in Figure 6's optimal performance band.

2.3.2 INSERT(x): amortized logarithmic cost

Insertion, extensively as it may have been analyzed by previous research [2] [4] [5], is a somewhat tedious process. Instead, we chose to experimentally verify that claim. We worked with 3 different data sets: 100 thousand, 1 million and 10 million integers, inserting one item at a time, while meticulously keeping tabs on every individual insertion and the

occurring subtree’s size that goes through rebalancing. Pat Morin mentions $\frac{2}{3}$ as his $ALPHA$ of choice [5], whereas Galperin used a handful of values: 0.55, 0.6, 0.65 [4]; various online sources also tend to pick values between 0.67 and 0.7. Since a SG tree with $ALPHA$ at 0.6 and below resembles an AVL in both shape and performance, we decided to pick 0.6667, which is close to all the aforementioned fractions. The results are illustrated in Figure 9:

- By far the most interesting finding is that the overwhelming majority of insertions in each data set - more than 99% - triggers a rebalancing operation of 0 size; essentially, the SG tree doesn't rebalance any subtrees at all. In fact, as the set size increases, that percentage approaches asymptotically 100%.
- The second most frequent subtree size is 4, at 0.05% of the insertions on average. As the subtree size increases, the frequency drops exponentially fast.
- Surprisingly, increasing the set size by orders of magnitude does not increase the maximum appearing subtree size: a literal handful of random outliers had more than 200 items to be rebalanced for both the 1M and 10M sets, the biggest of them being at around 450. Based on that, we can confidently assume that the occurring unbalanced subtrees are much smaller than the overall tree size and independent of the set size, therefore the rebalancing operation should take $O(1)$ worst-case real time.

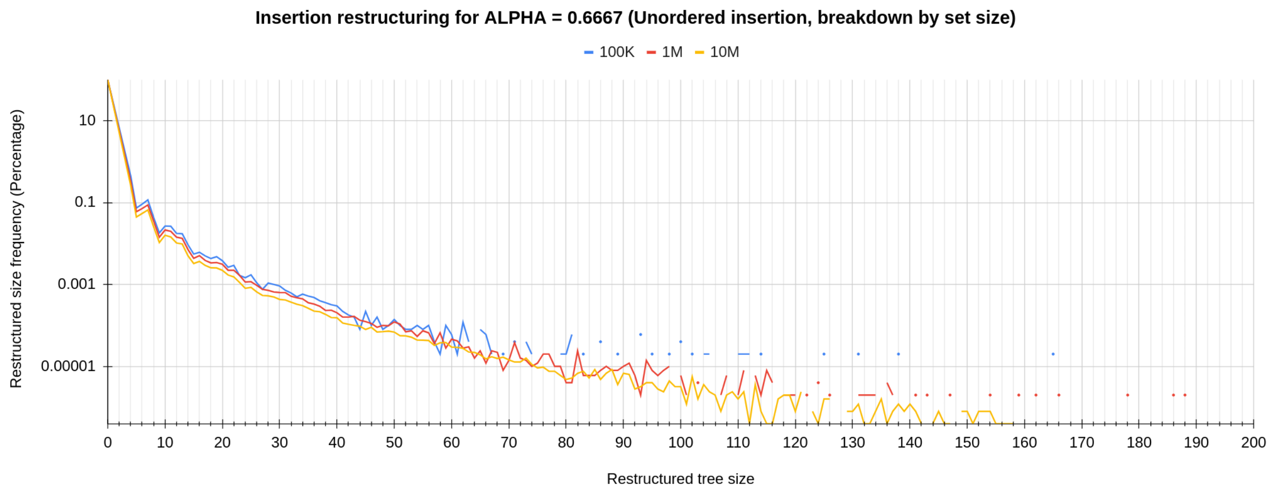

Figure 9. Rebalanced subtree size. At first, it was assumed that using bigger datasets should cause the frequency and size of outliers to increase. However, both frequency and maximal subtree size seem to be either independent of the dataset’s size or increase extremely slowly, possibly some derivative of $a(n)$ (inverse Ackermann function).

3. MULTI-OPERATION PERFORMANCE ASSESSMENT

3.1 Workload diversification: defining rules for the test

So far, we have only mentioned data regarding runs where a single operation type was used during the test cycle. However, it is often the case that deterministic tests in a controlled environment and real-time applications can bear wildly different results. We decided to conduct additional tests; this time, it wasn't just the integer input that was randomized, but also the operations themselves.

Essentially, we attempted to simulate a rudimentary DBMS session, consisting of a number of queries. During each query, it was given one integer at random with uniform distribution and one of 3 operations to apply upon the item: INSERT(x), REMOVE(x) or FIND(x). We then made the following assumptions:

- An item can be picked and inserted, removed or searched for multiple times.
- If the chosen item is not inserted, an INSERT operation is picked for it.
- If the chosen item is inserted, either FIND or REMOVE is picked at random.
- There exists a REMOVE/FIND bias; it causes one operation to be picked more frequently at the expense of the other.

Assumption no.1 is an obvious one: the same operations are often typically used on the same items multiple times; for instance, searching for the same item. Points 2 and 3 are somewhat more technical: since INSERT and REMOVE utilize binary search, when an item already exists and doesn't exist respectively, they fail and, because of the similar execution times, can be mistaken for FIND ones and thus skew the results. Finally, introducing a fluctuating REMOVE/FIND bias will help us ascertain which BST implementation is more appropriate, based on the workload.

3.2 Scaling and bias parameters

As previously, AVL and SG trees (0.6667 alpha) were tested as to their performance: they were given a 100K integer set and a query ratio; that is, a factor designating the session size, in relation to the set size. In other words, the total amount of operations during a session is given by the following simple formula:

$$\#\, of\ operations\ =\ set\, size\ \cdot\ query\, ratio$$

Moreover, we tested the integer set with increasingly larger query ratios and quantified our R/F bias, giving it a range between 0 and 1:

- At 0, REMOVE(x) completely dominates, preventing any FIND(x) from being used,
- At 1, roles are reversed; FIND(x) is given exclusive preference.
- At 0.5, the probability of one of them being picked is exactly 0.5.

3.3 Testing for small datasets and sessions

Having set all of the parameters and metrics, we proceeded with the experiment, as illustrated in Figure 10.

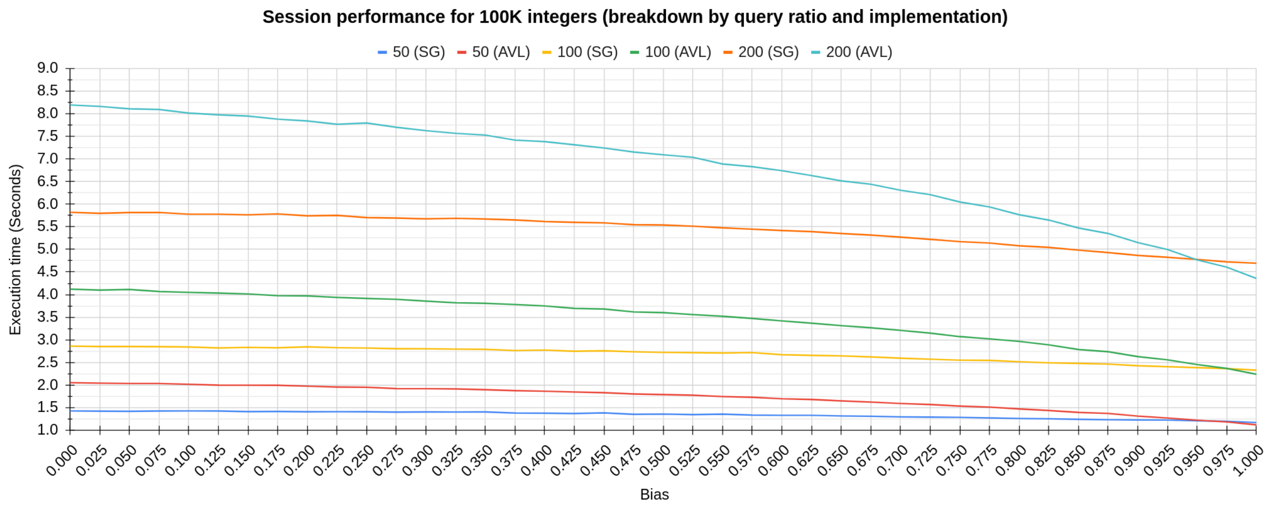

Figure 10. Session performance for 100K integers. Similar results occurred for lower query ratios as well, although differences in performance were negligible.

Our AVL implementation seems to severely lag behind the SG tree, which outperforms for every query ratio up to a point, after which AVL gains the upper hand. That is to be expected, since, given a bias close to 1, more searches lead to fewer removals chosen, which consequently lead to fewer insertions (remember assumption no.2 and 3).

3.4 Computational hurdles in bigger datasets and how to avoid them

Unfortunately, any attempt to increase set size and query ratio any further proved to be extremely time-consuming, due to exorbitantly impractical log linear execution times. It was thus decided to approach the problem from a different angle. Figure 11 shows the percentage of insertions, removals and searches in a session, given 2 different data sets and the same query ratio.

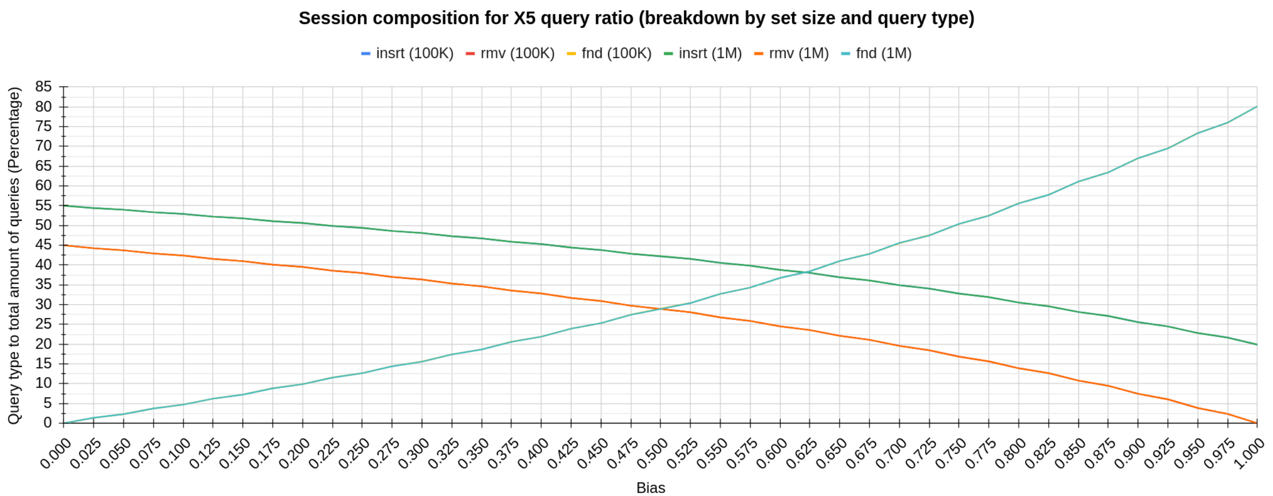

Figure 11. Session composition for x5 query ratio. Higher query ratios cause insertion and removal to asymptotically converge to the same percentage for a given bias.

As it turns out, 2 sessions with identical query ratios exhibit the same behavior, regardless of the set size. That clue significantly accelerated our experiments, since we can now use a small data set paired with an arbitrarily large query ratio and effectively simulate any session.

3.5 Results and performance assessment

We chose a 100-integer set and a 20K query ratio, shown in Figure 12 and 13.

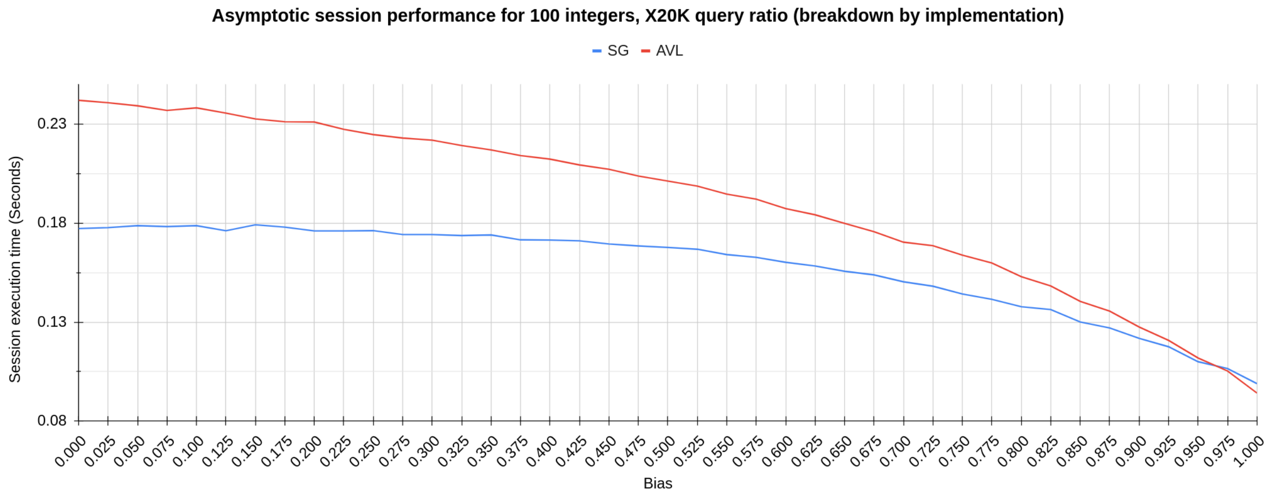

Figure 12. Asymptotic session performance. There is a clear downward trend in execution time, due to fewer time-consuming insertions and removals being picked, in favor of more FIND operations. Eventually, AVL becomes superior, since its structure is closer to a perfect tree.

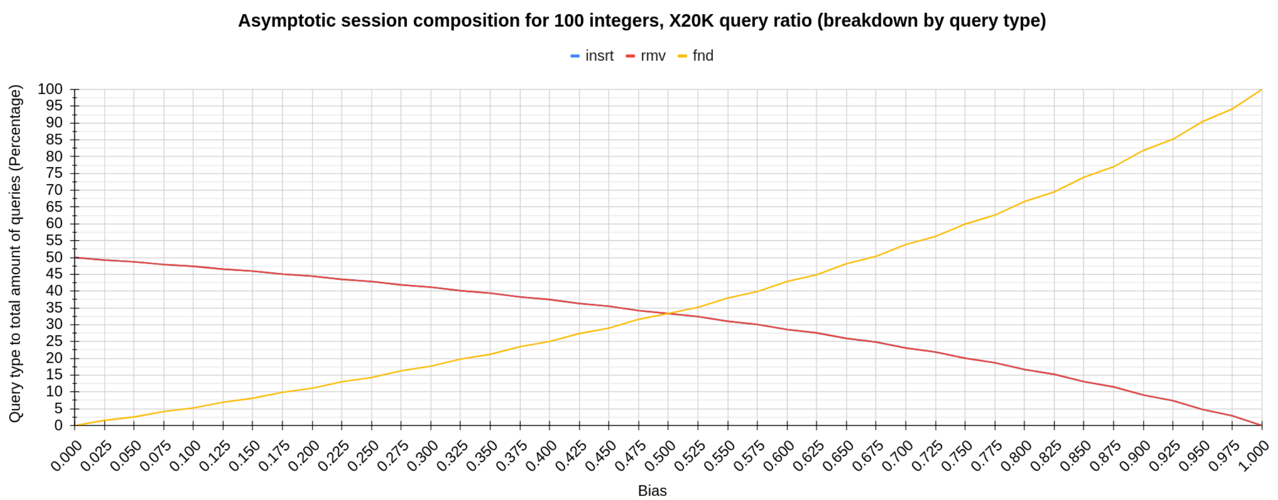

Figure 13. Asymptotic session composition. Insertions and removals follow the same curve; however, the number of insertions is ever so slightly higher than those of removals, since items have to already exist, in order for them to be removed. At 0.5 bias, the percentages of INSERT, REMOVE, FIND are asymptotically equal to each other, at 33.3%.

The SG tree outperforms AVL, even for extremely search-intensive workloads. More specifically, Figure 12 shows that the two performance curves intersect at a point between 0.95 and 0.975 bias, after which AVL becomes superior. With Figure 13's help, we can deduce that the point corresponds to about 90.5%; namely, 90.5% of the session's operations consists of FIND(x) and the remaining percentage splits evenly between INSERT(x) and REMOVE(x), at 4.75% each. It is safe to conclude that, unless one uses more than 90% of their time searching for items, switching to SG trees would be the more efficient choice.

4. DISCUSSION AND FUTURE WORK

For the purposes of our experiments, we used several techniques to optimize our SG tree implementation:

- The insertion path is stored in the stack, therefore the use of parent pointers becomes redundant.
- To avoid encumbering our nodes with additional information, such as subtree size, we used a recursive function to calculate the size of each node's tree, as well as its ancestors' in the insertion path, on the fly. Furthermore, since each node in the insertion path already has one child's size pre-calculated, we need only calculate the sibling's size, add the 2 children's sizes plus 1 and move up to the parent:

  $$size(path[x-1]) = size(path[x]) + size(sibling) + 1$$

  where $path[x]$ denotes the node in question and $path[x-1]$ its parent. At $path[0]$, we reach the root of the tree.
- To calculate the maximal permissible height/depth, one has to constantly resort to logarithms. However, since logarithms are transcendental in nature, utilizing them often incurs considerable time penalties, even in high-performance languages. We chose to use the Euler-Mascheroni approximation instead, and by doing so, we managed to reduce insertion times by about 4.5% on average.

However, we have yet to consider the following two optimization schemes, mainly due to them going beyond the scope of this Thesis:

- Parallelization for INSERT(x) and REMOVE(x): breaking the rebalancing load into multiple threads could improve performance for both operations. One could argue that insertions don't really trigger any meaningful rebalance; however, considering that any rebalance during removals incurs massive time penalties, there is still some performance boost to be had.
- Extension of the SG scheme to m-ary trees, effectively creating a loose alternative to B-trees. One should nonetheless take into account the fact that multiple siblings can potentially cause insertion times to deteriorate during rebalances, due to increasing times in recursively calculating subtree sizes. Another key difference is that $ALPHA$ in m-ary trees is bounded between $(\frac{1}{m}, 1)$. Thus, our BST implementation just so happens to be the special case, for which $m = 2$.

5. CONCLUSIONS

In our work, we extensively experimented with scapegoat trees -loosely balanced BSTs- whose absence of additional intranodal information makes for very attractive data structure alternatives for memory frugal applications, since only 3 global values are necessary at all times:

- $ALPHA$: a fraction, α for short, between (0.5, 1), sets an upper bound on the number of nodes a node, let it be A, can hold in its childrens' subtrees. Thus, each of A's children can't contain more than α·100% of the nodes in A's tree.
- n: the number of nodes in the entire tree
- q: a loose estimation of the maximal permissible height of the tree: $h_{max} = log_{1/a}q$, bounded between n and $2n$. If REMOVE(x) isn't implemented, our SG implementation can be simplified even further, only requiring 2 global values, that is, $ALPHA$ and n.

In a SG tree, a typical rebalancing operation begins at an external node, that is, the newly inserted node, and examines higher ancestors until a node (the "scapegoat") is found that is so unbalanced, that the entire subtree rooted at the scapegoat can be rebuilt at $O(n)$, making both insertions and removals' cost $O(logn)$ amortized time and $O(logn)$ worst-case real-time for searches.
Proving removal's amortized time is trivial, however the same can't be said for insertion. Therefore, we ran tests for multiple dataset sizes, in order to estimate the average subtree size. We found out that, as the dataset grows bigger, the amount of subtrees having size $O(1)$ tends asymptotically to 100%, and even the occasional outliers are very small and highly unlikely to appear.
The SG tree is more loosely constructed than an AVL, having a considerably bigger average height/depth. In spite of that, we decided to have the two BST implementations run 3 basic BST operations (INSERT(x), REMOVE(x), FIND(x)), 2 kinds of datasets (ordered, unordered) and 2 kinds of operation actions upon the dataset (same-order, shuffled) and compare their respective results.
The SG tree often achieved lower execution times than the AVL in multiple scenarios, for every basic BST operation that they were tested on:

- Insertion: SG performs in between AVL ordered and unordered set times, and, while not universally superior to AVL, its performance is more consistent and better than AVL's unordered insertion.
- Removal: SG massively outperforms AVL in both ordered and unordered sets, same-order and shuffled removals.
- Search: SG outperforms AVL in both ordered and unordered sets for same-order searches, for $ALPHA$ below 0.6. When it comes to shuffled searches, there's mixed performance, dependent on α. Again, given a sufficiently low α, SG has a performance advantage over AVL.

Overall, our implementation's competitive advantages compared to a typical AVL were thoroughly illustrated: in most scenarios and every basic BST operation, SG appears to have multiple strong points.
Additionally, we decided to check the SG tree's performance, by choosing one of each basic operation at random as well, instead of just an item, in order to simulate a real-time query session. We devised a number of rudimentary rules that would have to be applied at all times, such as REMOVAL(x) and/or FIND(x) not being chosen if the item picked isn't already in the tree and a REMOVE/FIND bias, which can influence the amount of REMOVE(x) and FIND(x) operations being picked during a query session.

We tested both implementations for different dataset sizes. However, due to the log linear execution times of the session, running them for bigger sizes proved problematic, therefore we resorted to alternative computation methods. We showed conclusive evidence that SG implementations perform better than AVL even for query sessions with close to 90% of them consisting of FIND(x) operations.
Finally, we describe the various optimization techniques that were implemented in our SG tree and we propose further potential improvements.

TABLE OF TERMINOLOGY

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Worst-case	χείριστη περίπτωση
INSERT	εισαγωγή
REMOVE	διαγραφή
FIND	εύρεση
Scapegoat	αποπομπαίος τράγος
Amortized	αποσβεσμένος

ABBREVIATIONS - ACRONYMS

BST	Binary Search Tree
AVL	Adelson-Velsky and Landis
SG	Scapegoat
R/F	REMOVE/FIND
K	Thousand
M	Million
DBMS	DataBase Management System

REFERENCES

[1] Adelson-Velsky, Georgy; Landis, Evgenii (1962). "An algorithm for the organization of information". *Proceedings of the USSR Academy of Sciences*, 1962.
[2] Galperin, Igal; Rivest, Ronald L., *Scapegoat trees*, *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*. Philadelphia: Society for Industrial and Applied Mathematics, 1993
[3] Andersson, Arne, *Improving partial rebuilding by using simple balance criteria*, Proc. Workshop on Algorithms and Data Structures, *Journal of Algorithms*. Springer-Verlag, 1989.
[4] Galperin, Igal, “Chapter 3 - Scapegoat trees”, *On Consulting a Set of Experts and Searching* (Ph.D. thesis), MIT, 1996.
[5] Morin, Pat., "Chapter 8 - Scapegoat Trees", *Open Data Structures*.

The scapegoat tree was developed in C and its repository is located at: https://github.com/vasilisvenieris/scapegoat_tree