SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2006; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Rehabilitation of an unloved child: semi-splaying

Brinkmann Gunnar[*,†], Degraer Jan, De Loof Karel

Department of Applied Mathematics and Computer Science, Ghent University, Krijgslaan 281–S9, B–9000 Ghent, Belgium

SUMMARY

Splay trees are widely considered as the classic examples of self-adjusting binary search trees and are part of probably every course on data structures and algorithms. Already in the first seminal paper on splay trees [7] alternative operations were introduced, among which *semi-splaying*. On the one hand the analysis of semi-splaying gives a smaller constant for the amortized complexity, however on the other hand the authors write: *Whether any version of semi-splaying is an improvement over splaying depends on the access sequence. Semi-splaying may be better when the access pattern is stable, but splaying adapts much faster to changes in usage.* Maybe this sentence was the reason that nobody seriously ran tests to compare the performance of semi-splaying and splaying. Semi-splaying is conceptually simpler than splaying, has the same asymptotic amortized complexity and, as will be clear from empirical data presented in this paper, the practical performance is better for almost all kinds of access sequences. So due to its efficiency there are good reasons to use semi-splaying instead of its more prominent brother for applications. Moreover, its simplicity makes it also very attractive for teaching.
Copyright © 2006 John Wiley & Sons, Ltd.

KEY WORDS: semi-splaying, splay trees, self-adjusting binary trees, experimental results

1. INTRODUCTION

Splay trees [7] are binary search trees that guarantee a good overall performance without storing any balancing information. Though in the worst case an access to a splay tree with n nodes may have linear cost, the amortized cost of an access is $O(\log n)$ — the same as in balanced search trees like AVL-trees, red-black-trees, etc. The same applies to all variants of semi-splaying considered in the remainder of this article. For all these variants the proof of

[*]Correspondence to: Gunnar Brinkmann, Department of Applied Mathematics and Computer Science, Ghent University, Krijgslaan 281–S9, B–9000 Ghent, Belgium.
[†]E-mail: Gunnar.Brinkmann@ugent.be

the amortized complexity is either already given in [7] or very similar to the proof given there, therefore the proofs will not be given here. The proof of the logarithmic amortized complexity of an access via semi-splaying is in fact just one case of the proof for splaying. Because the proof is shorter but still contains the same ideas, it is better suited to be used in teaching.

A large amount of work has been devoted to splaying. Some articles discuss theoretical aspects of splaying and variants of splaying (see e.g. [4], where splaying and variants of semi-splaying are compared theoretically), other articles compare the practical performance of splaying with other balancing techniques and modifications of splay trees for integer keys (e.g. [1, 2, 5, 6]) or more specialized data [8]. But although we checked quite a lot more articles than cited in the references, to our best knowledge and great surprise nobody seems to have compared the practical performance of splaying and semi-splaying. In [1, 6] modifications of splay trees are examined that do not splay on every access, but only with a fixed probability or for every k-th lookup for a given k. These modifications perform better than pure splaying. As our results will show, (pure) semi-splaying techniques already outperform splaying, so combining the randomized or partial techniques developed in these papers with semi-splaying is an interesting direction for future research, but we will not discuss it in this paper.

Some of the results outlined in earlier articles are contradictory. E.g., in [2] it is stated that top-down splaying is 3 times faster than bottom-up splaying while in [8] it is stated that top-down splaying is slower for these (not that special) purposes than bottom-up splaying. We can neither explain nor reproduce these results. As will be seen in the sequel, in our tests with two independent implementations top-down splaying is faster than bottom-up splaying, but the difference is less than 30% for all the cases we ran.

In [8] semi-splaying is also discussed. The authors write: *In all cases, we have found that semi-splaying performs worse than all other variants described in this paper, including RSTs. For this reason, we do not report detailed results here.* Though referring to [7], their explanation of semi-splaying makes us assume that they used another interpretation than the one described in [7]. They write that semi-splaying does not move the accessed node to the root, but only part of the way to the root. In fact semi-splaying promotes a node at least half of the way to the root, but sometimes also the whole way.

Since splay trees are part of probably every course on data structures, we will not describe the basic operations of splay trees here. The simplest version of semi-splaying which we will refer to as *lazy semi-splaying* can be described in a bottom-up way as follows: when adding or looking up a key in the tree, after having accessed the key, the path is traversed backwards and restructured. Initially the last node on the path is taken as the actual node. After having visited 3 nodes, this partial path of 3 nodes is replaced by a complete binary tree with 3 nodes. Then the root of this tree is taken as the actual node and we continue recursively until the remaining path contains only 1 or 2 nodes. The replacement of the paths is shown in Figure 1. Note that this differs slightly from the definition in [7] where a remaining path of length 2 is replaced by the other possible path of length 2 as shown in Figure 2. Leaving out this extra operation makes lazy semi-splaying even easier and — as our test shows — normally also increases the performance of the bottom-up version slightly. In Figure 5 the result of various ways how to proceed with a remaining path of length 2 are displayed. The best way to proceed in bottom-up semi-splaying in the case of an access path of odd length is to either replace

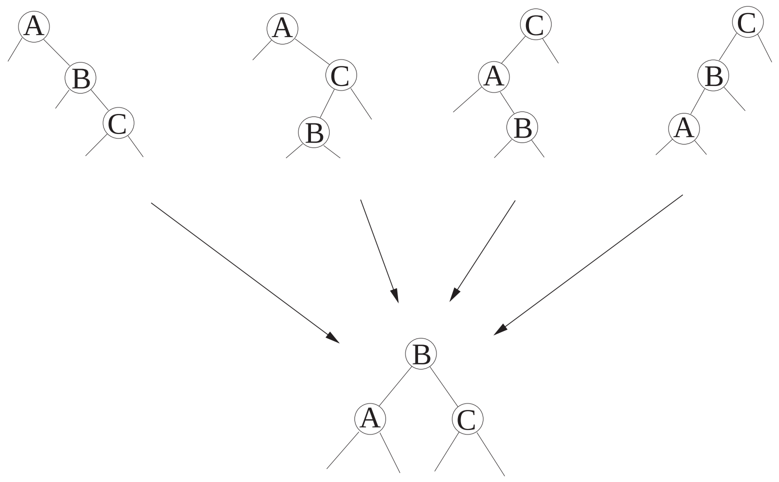

Figure 1. The replacement of a path with 3 nodes by a tree in semi-splaying.

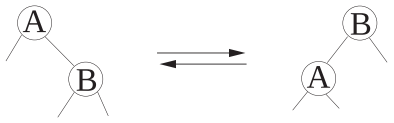

Figure 2. The replacement of a path with 2 nodes.

the 2-node-path at the bottom of the tree (this is also done by our top-down implementation following [7] and gives the best results for extremely biased access sequences) or to start at the bottom and leave the remaining 2-node-path unchanged, which performs best if the access sequence is not too biased. Note that the top-down implementation does not know the length of the access path when starting to splay, therefore it is not possible to place the remaining 2-node-path in the root.

So lazy semi-splaying performs some path compression of the access path with the effect that if the distance of the accessed node is d before the access, it is at most $\lceil d/2 \rceil$ after the access. In cases where the length of the access path is odd and every partial path with 3 nodes is of the form right-left or left-right, there is no difference from splaying and hence the accessed key is also splayed to the root.

The informal reasoning that convinced us of the usefulness of comparing semi-splaying techniques and splaying is as follows:

(i) As already noted in [7], the analysis of the amortized complexity gives a smaller constant for semi-splaying than for splaying. This suggests that for unbiased data semi-splaying may outperform splaying.

(ii) Semi-splaying also splays accessed nodes in the direction of the root, so keys that are looked up very often will end up close to the root. However, keys that are not that often looked up have a much smaller chance to be splayed right to the root and push the keys with high lookup frequency to lower levels. This may be an advantage in cases with a biased access sequence with a smooth probability distribution (like the *Zipf distribution* used here). With other words: in cases where you don't have a sharp boundary between some subset of keys that are looked up very often and with almost even probability inside the set and another subset of keys that are practically not looked up at all.

Argument (ii) lead us to design and test another variant of semi-splaying called *independent semi-splaying*. This variant can best be described in a top-down manner: starting at the root, the access path is split into non-overlapping sub-paths with 3 nodes each. Similar to semi-splaying, these sub-paths are replaced by complete binary trees with three nodes. In case of an independent sub-path with 2 nodes remaining, it turned out that especially for strongly biased access sequences it is a slight improvement to apply the operation in Figure 2. In Figure 3 examples are shown how the different kinds of splaying restructure an access path. In the tables and figures of the results section independent semi-splaying stands for a top-down implementation with sub-paths with 2 nodes also replaced.

The main benefits of independent semi-splaying are a smaller amount of restructuring of the tree and the fact that nodes with a small access frequency can not be promoted to the root or close to the root in one step. A disadvantage (which also turns up as a larger constant in the analysis of the amortized complexity) is that the access path is compressed by a factor of only 2/3 instead of 1/2. Tests were necessary to show which of these effects has a stronger influence on the performance.

Sleator and Tarjan argument that semi-splaying will perform worse in the case of varying access patterns due to the fact that it normally promotes keys more slowly to the root than splaying. This is obviously true when one considers the case of an access pattern that almost always looks up the same node a few times and then switches to another distribution. On the other hand, when increasing the number of changes of the access sequence for a given biasedness and a given number of accesses, the limit is a non-biased distribution. So we also examined the behaviour of semi-splaying and splaying for an increasing number of changes in the access sequence.

2. TESTING

There are various ways how to produce biased input data. In this article we will use the generalized Zipf distribution described in [2]. The biasedness for various parameters from

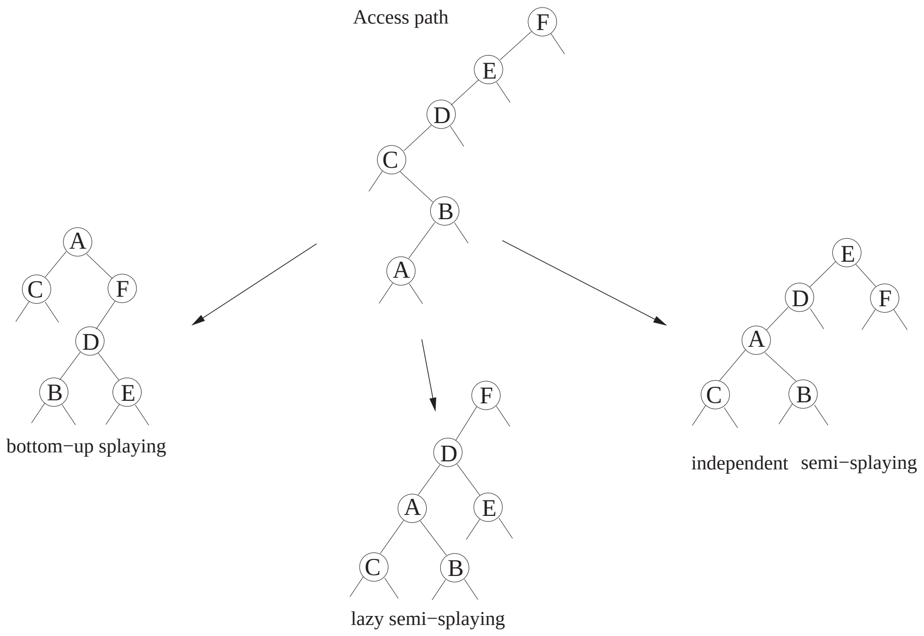

Figure 3. An example for the restructuring of an access path by three different ways of splaying.

completely unbiased (Zipf parameter 0) to the most active 1% of the keys being looked up 99% of the time (Zipf parameter 1.4 for 5.000.000 keys) is shown in Figure 4. For fixed parameters the biasedness grows slowly with the number of keys. For 5.000.000 keys the parameter 0.9 produces a distrubution approximately following the often cited 80-20 rule where the most active 20% of the keys are looked up 80% of the time. We also ran tests with other distributions of the access patterns, e.g. the probability distributed according to the n-th root (with the parameter n describing the biasedness) and a biased distribution where all the elements had one out of 2 possible probabilities to be looked up. When choosing the corresponding parameters in a way that an approximately even biased access pattern is generated, the results differed only very slightly.

When running tests with l lookup operations on trees with n keys, we first filled an array of length n with keys $1, \ldots, n$. Then we applied a random permutation to this array and added the keys in the new order to the tree. Afterwards another random permutation was applied and l times a key was chosen for lookup with the probability given by the generalized Zipf distribution. When testing the effect of varying access patterns, additional

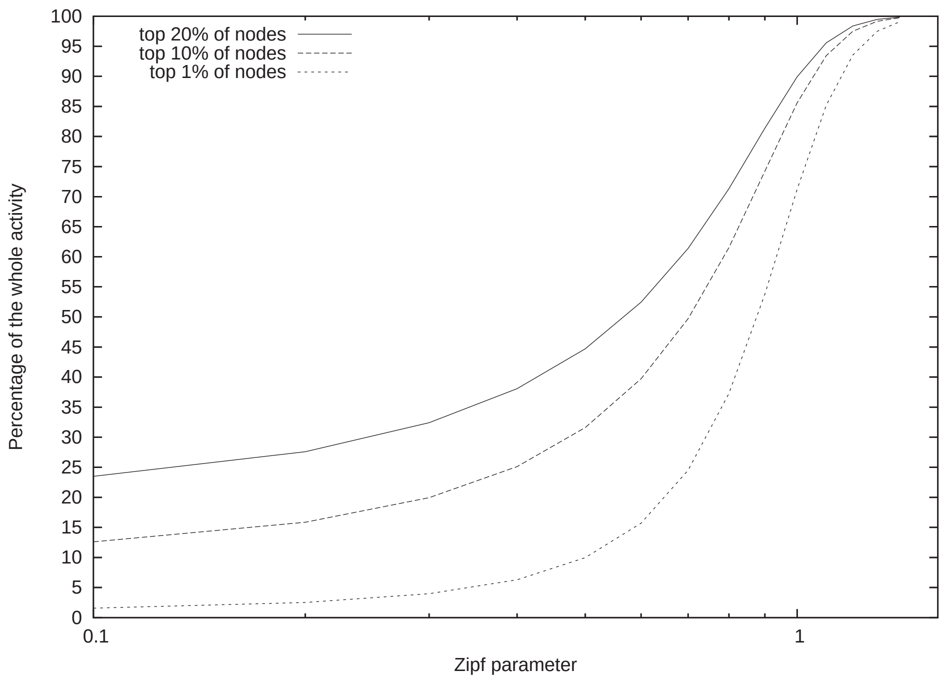

Figure 4. The percentage of lookup operations used for the most active 1, 10 and 20% of 5.000.000 nodes as a function of the Zipf parameter.

random permutations were applied to the array after a given number of steps. In order to avoid programming errors for the data-generator we tested the output of the routine for showing the predicted distribution.

We also ran some tests including remove operations, but because the results were very similar to those presented here and did not seem to provide any new insight, we did not implement this operation for all variants. The fact that splay trees are especially interesting for biased data and that in case of a large ratio of add/remove operations the data tends to be unbiased is another argument to restrict the attention to lookup operations. Nevertheless it should be said that for the efficiency of the remove operation in splaying it is important not to use the often proposed method that first splays the element that must be removed to the root. This is an elegant but obviously not efficient way. When using this technique and a lot of remove operations, the efficiency of splaying relative to semi-splaying decreased. It is more efficient to remove nodes in splay trees in the same way as we did for semi-splaying and as is done in binary trees. First we looked up the node to be removed. In case this node is a leaf or has only one child, it can be easily removed (possibly attaching a child to the parent). In case of two children we proceeded to the node with the biggest key in the left subtree or the one with the smallest key in the right subtree. Then we removed this node placing its key in the node with the key that is to be removed. The path along which splaying or semi-splaying was applied is the access path to the parent of the removed node.

It is quite difficult to compare running times. The results depend a lot on the actual implementation, the type of key that is stored in the nodes and even the compiler. Changing any of these could not only result in different absolute running times, but also in different relative running times. In cases where the comparison of two keys is the most expensive part, it all comes down to counting comparisons or nodes visited during the operations. In cases where the comparisons are cheap (like in our case where integers are compared) other effects including implementational details become more important. We tried to implement all methods with the same care, but can't make sure that one or the other method would possibly allow more optimizations than the other. Therefore we will just give two figures comparing the running times and in the others concentrate on the implementation independent value of the number of nodes visited.

To make sure that no programming errors influence the results, we implemented all methods twice — once in C and once in Java. Comparing the number of nodes visited for a given input gives a good test that actually the same trees are built.

3. RESULTS

In this section we will give the empirical results of our tests. These tests were run on an initial tree with 5.000.000 nodes on which 500.000.000 lookup operations were performed.

Figure 5 displays the effect of the different ways to deal with remaining paths of length 2 in semi-splaying. The results are given in number of nodes visited during the lookup operations relative to the number of nodes visited by lazy semi-splaying. One can easily observe that except in extremely biased situations it is a good choice not to do any 2-node-path operations and choose the unmodified 2-node-path at the root. In extremely biased situations switching at the end of the path (like also done by the top-down implementation) is optimal.

In Figure 6 and Figure 7 the running times of the C-, resp. Java-programs relative to the time needed by the implementation of top-down splaying — the faster one of the two traditional splaying techniques — are given. The times include reading 40 bits of input data per operation. The data is of type (x,n) from the input stream (where x is an 8 bit character denoting whether to add or look up the key n which is a 32 bit integer) together with the time needed for building the tree and looking up the items. In both cases it turns out that (as expected) top-down implementations have an advantage over bottom-up implementations and that independent semi-splaying is the fastest of these methods for unbiased input as well as for extremely biased input. In order to minimize the effects of external influences (like other programs using the cache) and the seed used for the data generation, the results given are the average of 3 runs with 3 different seeds for the data generation program.

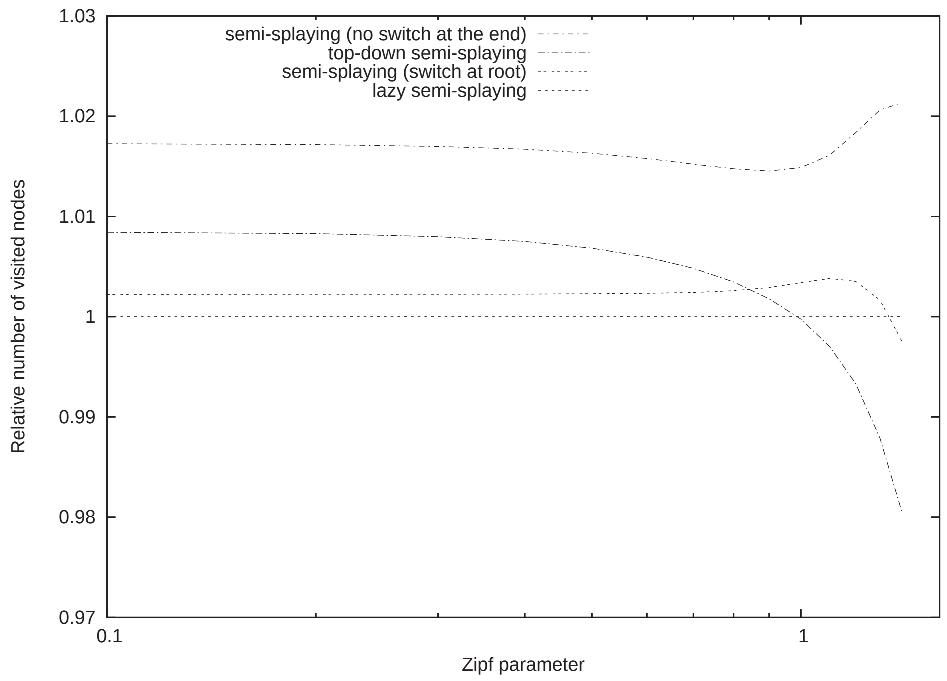

Figure 5. The effect of different ways to deal with paths of odd length by semi-splaying variants relative to lazy semi-splaying.

In Figure 8 the numbers of nodes visited during the lookup operations by lazy semi-splaying, top-down semi-splaying and independent semi-splaying are shown relative to the number of nodes visited by top-down splaying (which was chosen as reference because it performs slightly better than bottom-up splaying). The numbers are averages over 3 runs with different seeds, The effect of the seed was extremely small for unbiased sequences (less than 0.01% difference in the number of vistited nodes) and reached a maximum of 3% for extremely biased access sequences (Zipf parameter 1.4). One can see that all variants of semi-splaying perform better than usual splaying.

In case of small trees, for a given ratio of $k\%$ of the nodes, the average path length from the root to those $k\%$ of the nodes that are closest to the top compared to the average path length of all nodes is larger than the ratio in big trees. This can lead to a very slightly different ratio of visited notes as can be seen in Figure 9. The overall result that semi-splaying performs better than splaying is nevertheless as clear as in the case of larger trees.

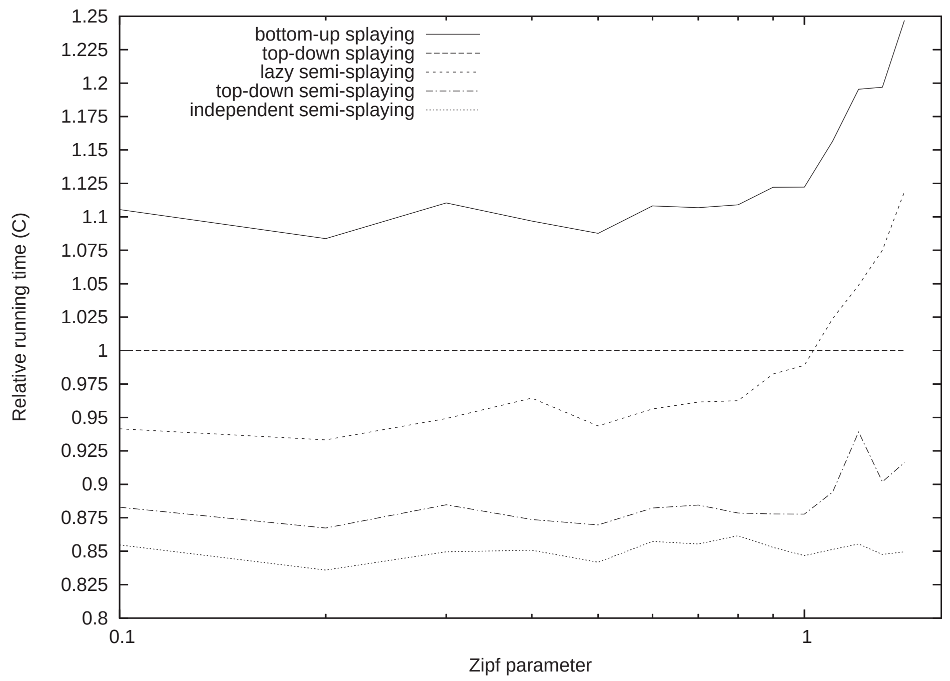

Figure 6. The running times of the C implementations of some splaying methods relative to top-down splaying.

In Figure 10 to Figure 12 the effect of changing access patterns is shown. To obtain these results the program generating the access sequence regularly applied a random permutation to the array with the keys. Each permutation results in a completely new access sequence. For the first two figures the permutation was first applied only once (so the length of the access sequence is 100 times the size of the search tree), then every 50.000.000 lookup operations, every 5.000.000 lookup operations, etc., to finally every 5.000 lookup operations for Zipf parameter 0.9 and only 500 lookup operations for Zipf parameter 1.3 (which is 0.01% of the size of the search tree). Because the effect of changing access patterns is much stronger for strongly biased access patterns, we display the results for Zipf parameters 0.9 (which gives a distribution that is somewhat more biased than the often cited 80-20 rule) and 1.3 which gives an extremely biased distribution with the most active 1% of the keys having a lookup probability of more than 97%. The results show that although the tendency predicted in [7] is clear and for extremely fast changing and very strongly biased access patterns splaying has an advantage, it is questionable whether such extreme conditions do occur in practical applications.

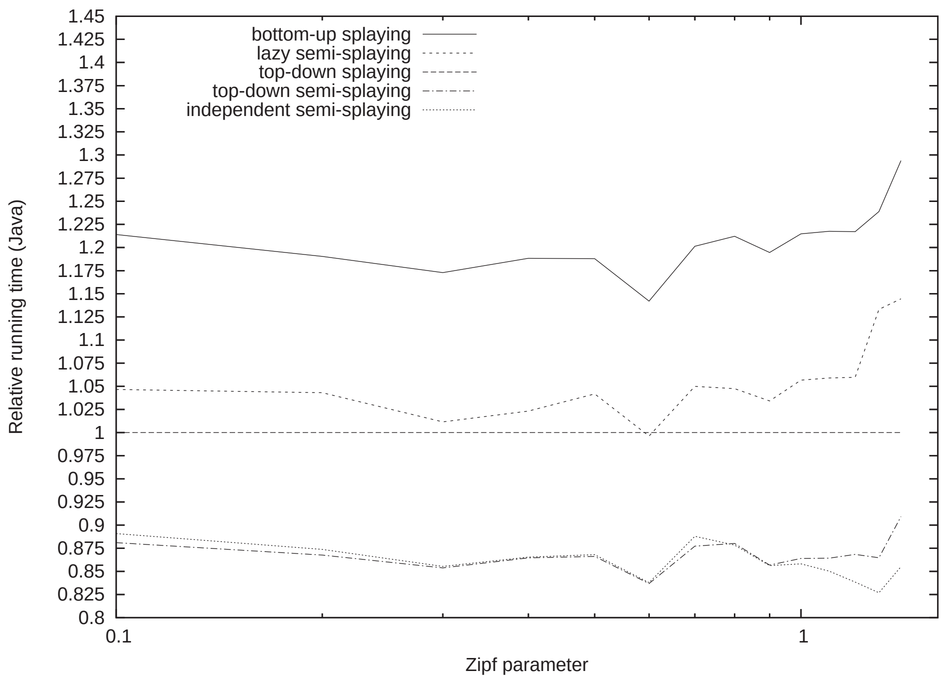

Figure 7. The running times of the Java implementations of some splaying methods relative to top-down splaying.

Because a further increase of the number of changes leads to a non-biased situation when after every acces the access pattern is changed, it is clear that for even faster changing sequences semi-splaying methods will be better again. This phenomenon is displayed in Figure 12. For the tests displayed in this figure, after every lookup operation the access pattern is changed with a certain probability starting at 0.001 and ending with 1 which is equivalent to a non-biased distribution. Because it is very expensive to generate the input data in cases where the access pattern is changed very often, these tests were run with less lookup operations and on smaller trees than before. The figure displays the results of 10.000.000 lookup operations on a tree of size 200.000 and biased access patterns with Zipf parameter 1.4.

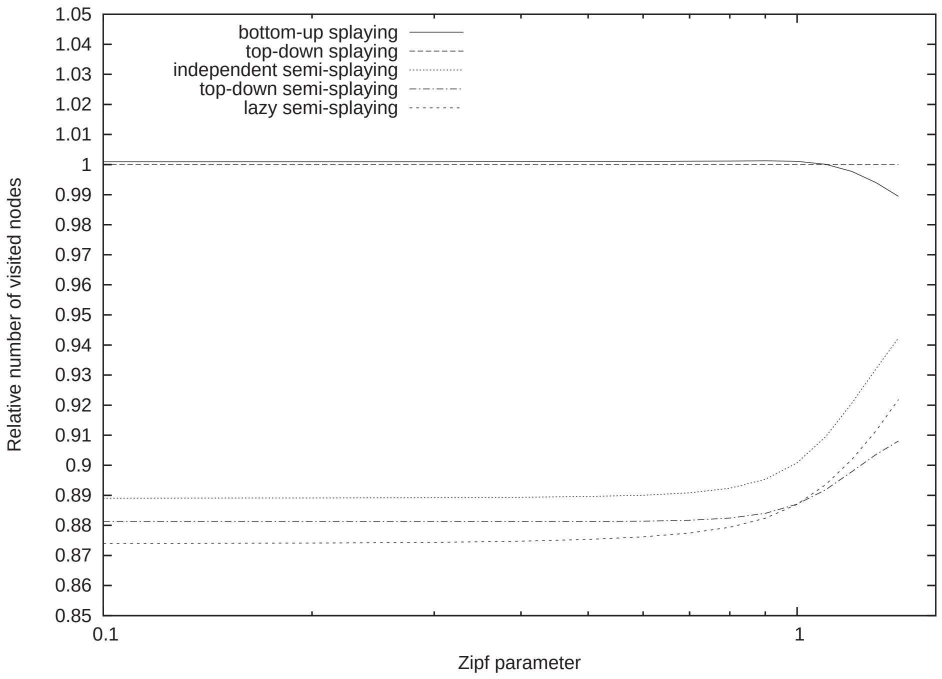

Figure 8. The numbers of nodes visited during the lookup operations of some splaying methods relative to top-down splaying.

4. CONCLUSION

The results show that semi-splaying, which was introduced in the same paper [7] as splaying, performs better than splaying under almost all possible conditions. This makes semi-splaying a good alternative for all applications where normally splaying would be applied. The reason why splaying became so prominent while semi-splaying is relatively unknown and much less studied is hard to understand.

Furthermore the concept of semi-splaying — especially the lazy semi-splaying variant — is much easier. Because also the proof of the average amortized logarithmic complexity is much shorter while still illustrating the same ideas, semi-splaying is also very well suited for teaching purposes.

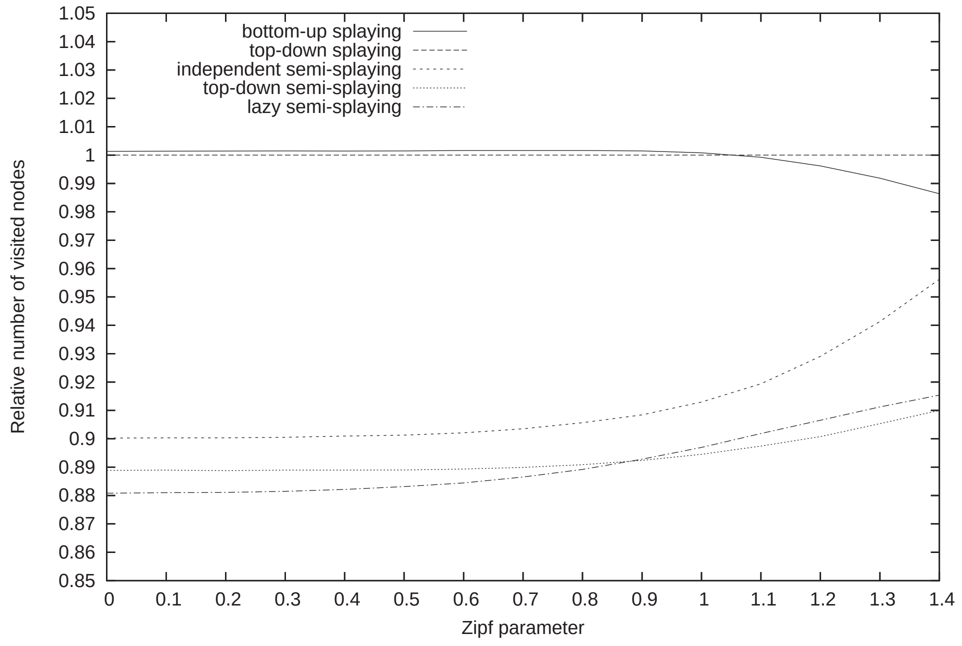

Figure 9. The numbers of nodes visited during the lookup operations of some splaying methods relative to top-down splaying in a tree of only 50.000 nodes with 5.000.000 lookup operations. Note that in this case the Zipf parameters correspond to a slightly less biased sequence than in the case of 5.000.000 nodes. E.g. in the case of parameter 1.4, the top 1% of the nodes have a chance of 94% to be looked up instead of 99%.

REFERENCES

1. Albers S, Karpinski M. Randomized Splay Trees: Theoretical and Experimental Results. *Information Processing Letters* 2002; **81**(4):213–221.
2. Bell J, Gupta G. An evaluation of self-adjusting binary search tree techniques. *Software—Practice and Experience* 1993; **23**(4):369–382.
3. Bell T, Kulp D. Longest-match string searching for Ziv-Lempel compression. *Software—Practice and Experience* 1993; **23**(7):757–771.
4. Fürer M. Randomized splay trees. *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms* 1999; Baltimore, Maryland, 17-19 January.
5. Klostermeyer W. Optimizing searching with selfadjusting trees. *Journal of Information & Optimization Sciences* 1992; **13**(1): 85–95.
6. Lee E.K., Martel C.U. When to use splay trees. *Software—Practice and Experience* Published Online: 5 Apr 2007;
7. Sleator D, Tarjan R. Self-adjusting binary search trees. *Journal of the Association for Computing Machinery* 1985; **32**(3):652–686.
8. Williams H, Zobel J, Heinze S. Self-adjusting trees in practice for large text collections. *Software—Practice and Experience* 2001; **31**:925–939.

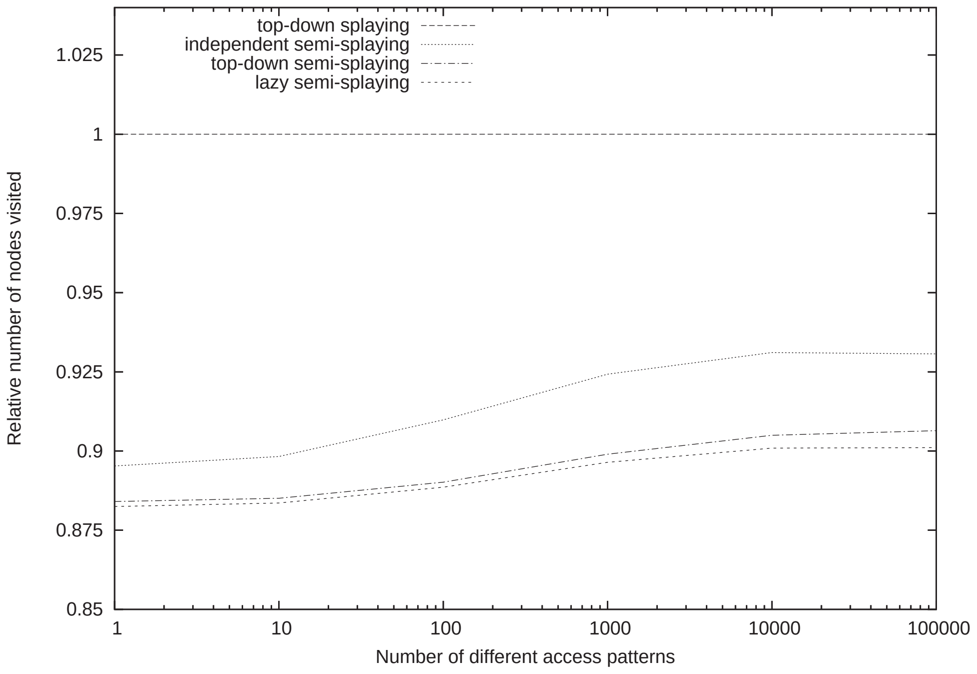

Figure 10. The development of the number of nodes visited during the lookup operations by some splaying methods for Zipf parameter 0.9 and changing access patterns relative to top-down splaying.

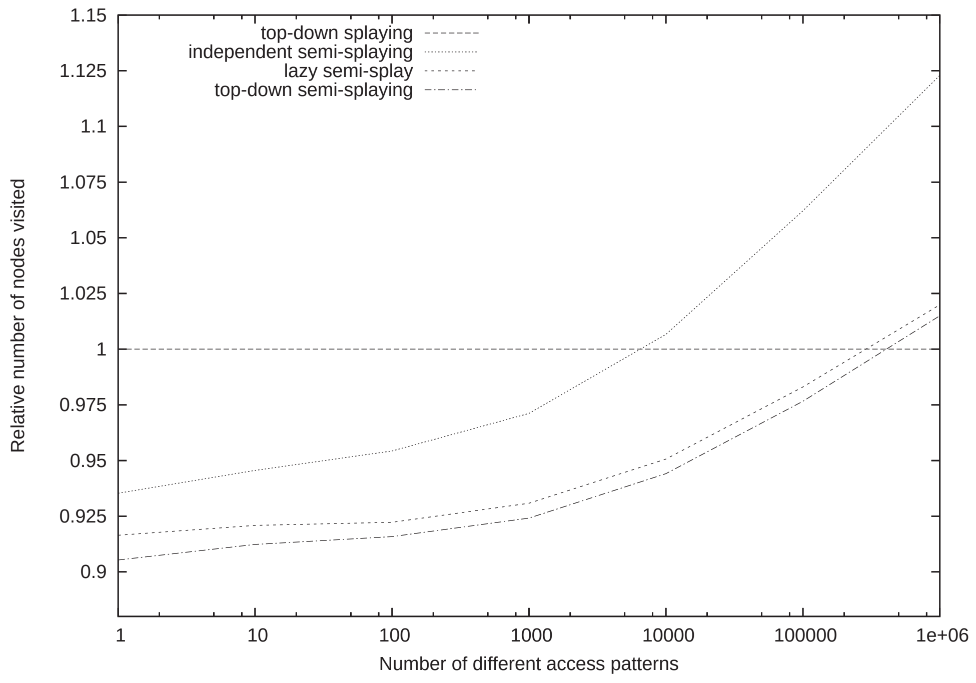

Figure 11. The development of the number of nodes visited during the lookup operations by some splaying methods for Zipf parameter 1.3 and changing access patterns relative to top-down splaying.

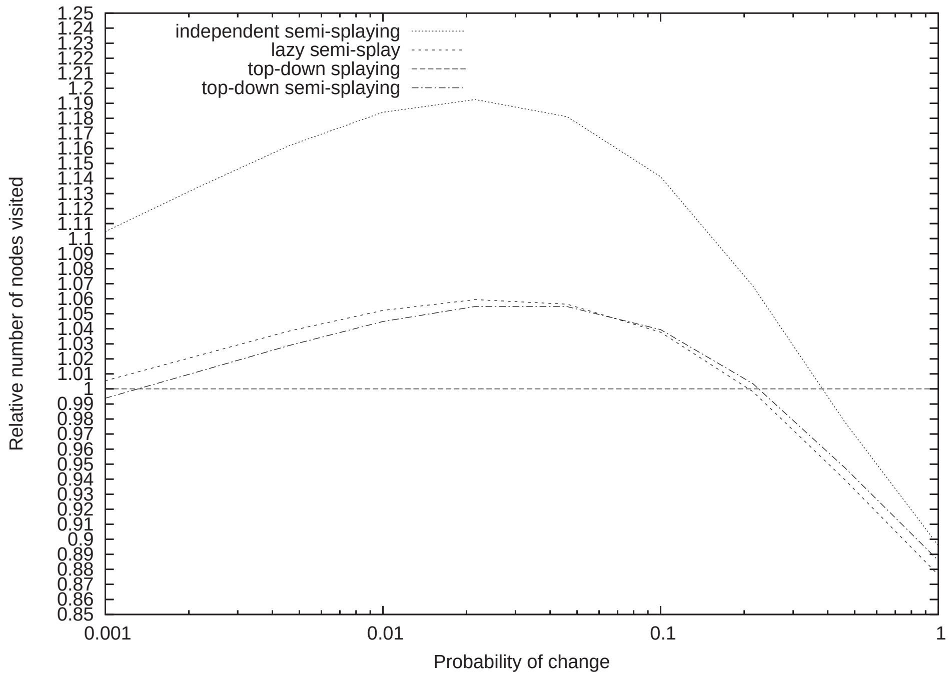

Figure 12. The development of the number of nodes visited during the lookup operations by some splaying methods for Zipf parameter 1.4 and very fast changing access patterns relative to top-down splaying.