Reducing Splaying by Taking Advantage of Working Sets

Timo Aho, Tapio Elomaa, and Jussi Kujala

Department of Software Systems, Tampere University of Technology
P.O. Box 553 (Korkeakoulunkatu 1), FI-33101 Tampere, Finland
{timo.aho,tapio.elomaa,jussi.kujala}@tut.fi

Abstract. Access requests to keys stored into a data structure often exhibit locality of reference in practice. Such a regularity can be modeled, e.g., by working sets. In this paper we study to what extent can the existence of working sets be taken advantage of in splay trees. In order to reduce the number of costly splay operations we monitor for information on the current working set and its change. We introduce a simple algorithm which attempts to splay only when necessary. Under worst-case analysis the algorithm guarantees an amortized logarithmic bound. In empirical experiments it is 5% more efficient than randomized splay trees and at most 10% more efficient than the original splay tree. We also briefly analyze the usefulness of the commonly-used Zipf's distribution as a general model of locality of reference.

1 Introduction

Many search trees facilitate efficient access to the stored items by keeping the tree in balance using rotations [1]. The balance invariant is maintained independent of the sequence of access requests observed. Splay trees [2], on the other hand, manage to do without any invariant, but need to splay in connection of each access and update. Splay trees lose the provably logarithmic worst-case bounds of individual operations, but still behave well under amortized analysis. The need for (expensive) splaying can be reduced by randomizing the decision of whether to splay or not in connection of an operation [3,4] as well as by heuristic limit-splaying algorithms [2,5,6].

Several theoretical results indicate that splay trees should work particularly well when there is locality of reference in the request sequence [2]. However, some empirical studies [6,7,8] have indicated that they could be actually at their best in highly dynamic environments, where the focus of locality drifts over time. Moreover, despite careful implementation basic splay tree variations have empirically been observed to be less efficient than red-black trees (RBTs), standard binary search trees (BSTs), and hashing at least in some situations [6,7]. Randomized adaptive data structures can do better [4,6], but only heuristic limit-splaying has been competitive in practice [6]. However, some recent studies [9,10] have demonstrated that in some settings splay trees may be more efficient

C.C. McGeoch (Ed.): WEA 2008, LNCS 5038, pp. 1–13, 2008.

than other BSTs. We hope that paying better attention to the properties of the input — splaying only when necessary and useful — would lead to more efficient performance.

Randomized splay trees achieve some practical savings without giving in on asymptotic efficiency. Nevertheless, they still do not pay any attention to the properties of the request sequence. The best advantage of randomization has been shown for fixed probability distributions, while request sequences with high dynamic locality of reference benefit from randomization only slightly [4]. Our aim in this paper is to study whether the amount of splay operations could be reduced more efficiently. In other words, we investigate how large savings (if any) can be achieved by monitoring the actual input that is seen.

In particular, we examine access request sequences that exhibit locality of reference in the form of *working sets* [11]. This means that, at any time interval, most accesses refer only to a small portion of all keys — the current working set. Real-world situations often conform with this assumption. Of course, we cannot afford to implement a too complicated request sequence monitoring method, because then we would be destined to lose in time consumption to the in-practice efficient (randomized) splay trees.

We will introduce and analyze a quite straightforward version of the splay tree which takes the existence of working sets in the request sequence into account. The algorithm maintains a (discounted) counter to monitor the (average) depths of recent searches in the tree. Low average search depth indicates that a working set exists near the root of the tree and is being actively used. Occasional deep searches do not change the situation. Only when the searches are constantly deep, is there need to update the splay tree.

The remainder of this paper is organized as follows. In Section 2 we briefly recapitulate splay trees and their relation to working sets. Section 3 presents the main idea of conditional adaptation of a binary search tree studied in this paper. The splay tree algorithm based on this idea is introduced in Section 4. An empirical evaluation of the algorithm is reported and analyzed in Sections 5 and 6. Finally, Section 7 gives the concluding remarks of the paper.

2 Adaptive Data Structures and Working Sets

A splay tree is a BST with the keys in symmetric order. In it the accessed item is elevated to the root of the tree using splay rotations [2]. These operations keep the tree pretty well in balance. Because no other balancing is enforced, a splay tree does not contain any additional information and, thus, does not require extra storage space. Because of its strategy, a splay tree also keeps recently accessed items very near the root. Thus, it automatically handles also working sets quite efficiently. On the downside, the accessed key is splayed to the root even if it is accessed only once during the whole access sequence. Also unnecessary splaying is executed even if the current working set is near the root and, thus, already efficiently accessible.

Sleator and Tarjan [2] proved many interesting bounds and properties for the time consumption of splay trees. In this paper we need only one of these results. The formulation of the following theorem comes from Albers and Karpinski [4]. Throughout this paper we denote by m the number of access requests in the operation sequence and by n the number of keys mentioned (all of them stored in the tree).

Theorem 1 (Balance Theorem). *The total access time incurred by an access sequence is at most* $3m \log_2 n + m + C$*, where* $C = n \log_2 n$*.*

There are two basic implementations for splay trees: top-down and bottom-up splaying. Asymptotically their access times are the same, but the practical efficiency of these implementations has been under some controversy [6]. In our experiments top-down splaying was always more efficient than the bottom-up version. Therefore, we report our results using top-down splaying with all the algorithms.

An interesting theoretical examination of splay trees has been presented by Subramanian [12]. He proposed a more general group of trees possessing similar properties as splay trees. The splay heuristic has additionally been applied in other tree structures [13]. Iacono [14] has also discussed the topic of splay trees and working sets. He presented new distribution sensitive data structures consisting of multiple trees and proved very interesting features for them.

After the submission of this paper we learned that independent of us Lee and Martel [10] have proposed an algorithm very similar to ours for cache efficient splaying. Their algorithm uses a sliding window of accesses. The algorithm executes splaying if a too large portion of the accesses is deeper than a predefined limit depth. They also present experiments somewhat similar to ours. However, their test setting is more static.

For self-organizing lists a heuristic with some reminiscence with the splaying operation is the move-to-front rule [15]. Amer and Oommen [16] have recently examined the effect of locality of reference in self-organizing lists.

3 Conditional Adaptation of a Binary Search Tree

Before introducing the BST algorithm intended to cope with working sets, we give the basic philosophy of the algorithm: how to identify a working set and its change.

We execute splay operations only when the active working set is not near the root or when the whole tree is unbalanced. Thus, unnecessary splaying is avoided in accessing a single item outside the working set as well as an item of the working set already near the root. The latter is useful especially when the access operations are approximately uniformly distributed among the items in a working set; in other words, there are not several layers of working sets. However, assuming this would be very restrictive and, thus, we do not use it in the paper.

In order to gain knowledge of the input we simply maintain a discounted depth counter, which gives us information about the relation between on-going access operations and the current working set. If the last few access operations are deep, we can conclude that the working set has changed or the tree is not in balance. Thus, there is need to splay to correct the situation.

For our counter we need an approximate value for the size of working sets w. See e.g. [11,17] for techniques on approximating w. With this value we can calculate the limit depth $limit_w$, which represents the acceptable average depth for access operations in working sets. We set $limit_w = a \log_2(w + 1)$, where the multiplier a is a constant chosen suitably for the current environment.

The value of the *condition counter* is updated in connection of an access operation to *depth* as follows:

$$counter \leftarrow d \cdot counter + depth - limit_w,$$

where the *discounting factor* d, $0 \leq d < 1$, is a constant regulating the impact of the history of access operations on the current value. The difference $depth - limit_w$ tells us how much (if at all) below the limit depth we have reached. If the value of *counter* is non-positive we may assume that splaying is not required. On the other hand, a positive value suggests that the operation is needed.

Taking discounted history into account ensures that isolated accesses outside the working set do not restructure the tree needlessly. On the other hand, giving too much weight to earlier accesses makes the data structure slow to react to changes in the working set.

We could as well let the value of $limit_w$ change during the execution of algorithm. In fact, in our empirical evaluation we use a dynamically changing $limit_w$. However, for a more straightforward analysis we assume for the time being that the value is constant. We also assume that the accessed item can be found in the tree; if not, the value of *counter* should be left unchanged.

There are alternatives for our approach. We could, for example, get rid of the whole discounting philosophy and use individual access counters for the keys [5]. These counters should be included either in the nodes of the tree or kept in a separate data structure. However, e.g., Lai and Wood [5] have already inspected the first approach with splay tree. Also Seidel and Aragon [18] introduced randomized search trees and Cheetham et al. [19] conditional rotating based on this approach. Because keeping the nodes free from additional information is an essential part of splay trees, we would nevertheless like to find another way.

On the other hand, if the counters were kept in a separate data structure, updating this information would be problematic. We are, anyway, mostly interested in only the last access operations. Thus, too static counters would not react quickly enough to the altering working set. Hence, we should have a technique to decrease the significance of old access operations. We are not aware of a solution that would not raise the running time too much. Lee and Martel [10] solve the problem by counting only the amount of deep accesses. Thus their solution loses information about the access depths.

procedure WSPLAY(x)

```
(1) if counter > 0 then
(2)     depth ← SPLAY(x)
(3)     counter ← depth − limit_w
    else
(4)     depth ← BSTACCESS(x)
(5)     counter ← counter · d + depth − limit_w
(6)     if counter > 0 then
(7)         SPLAY(x)
(8)         counter ← depth − limit_w
```

Algorithm 1. The WSPLAY algorithm

4 The Algorithm Detecting Working Sets: Wsplay

Let us now introduce an algorithm based on the information collecting approach described above. WSPLAY (Algorithm 1) simply splays whenever the condition counter implies that the working set is changing.

Function BSTACCESS implements the standard BST access and returns the depth of accessed item. As we want to execute the more efficient top-down version of splaying, we splay if *counter* indicates the need of splaying in the beginning of access. This is the case when the previous access operation — which necessarily included splaying — was deep. Thus we avoid doing unnecessary BST access before every top-down splay. Otherwise we access the item, update the condition counter, and splay if the updated counter value indicates a need for it. Observe that after each execution of the SPLAY function, the history of access depths is erased.

We assume that the function SPLAY also returns the original depth of the accessed item. Note that setting $d = 0$ makes WSPLAY very similar to the algorithm introduced by Sleator and Tarjan in the Long Splay Theorem [2, Theorem 7].

We now prove a logarithmic bound for the running time of WSPLAY. Recall that m is the length of the access sequence and n the number of nodes in the splay tree. The sequence of access operations H is divided in two disjoint categories. Let H_s consist of access operations including splaying and H_n of those without a splay operation. Because the tree structure does not change as a result of the accesses in H_n, the time consumption of these two sequences can be analyzed separately.

The sequence H_s essentially consists of splay operations. Only constant time overhead is caused by counter updating. Thus, the time consumption on H_s is bounded by Theorem 1.

To derive a bound for the time consumption on the sequence H_n, we analyze the values of variable *counter* during the access operations of a single continuous sequence $H_c \subseteq H_n$, $H_c = \langle h_1, h_2, \ldots, h_c \rangle$. Either the first access operation h_1 is the first access in the whole sequence H or its predecessor includes splaying. During H_c splay operations are not executed. Let us denote the values of the

variable *counter* right after the first update in the algorithm on line 5 with subindices, respectively. Note that the updates on lines 3 and 8 are not executed because no splay operations are executed in H_n. In particular, $counter_0$ is the value of the variable in the beginning of the access h_1. By the definition of H_c $counter_0 \leq 0$. Let $depth(h_i)$ be the depth of access operation h_i.

With these definitions we give a bound for the average access depth in the sequence H_c. To achieve this, we need to prove that if splay operations are not done the value of $counter_i$ gives a sort of a bound for the depth of access operation h_i. Intuitively the idea is to show that if the *counter* is never above 0, the average depth of accesses cannot be too much larger than $limit_w$.

Lemma 1. *If for all* l, $0 \leq l < c$, $counter_l \leq 0$, *then for all* i, $1 \leq i \leq c$,

$$\sum_{j=1}^{i}(depth(h_j) - limit_w) + counter_0 \leq counter_i.$$

Proof. During the access operations h_i, $1 \leq i \leq c$, no splay operation is executed and, thus, it holds that

$$counter_i = \sum_{j=1}^{i} (depth(h_j) - limit_w)\, d^{i-j} + d^i\, counter_0.$$

We prove the claim by induction over the index i.

Let $i = 1$. Because $0 \leq d < 1$, it is clear that $depth(h_1) - limit_w + counter_0 \leq depth(h_1) - limit_w + d \cdot counter_0 = counter_1$. Hence, the claim holds in this case.

Let us then assume that the claim holds when $1 \leq i = k < c$. We focus on the situation $i = k + 1$. By assumption we know that $counter_k \leq 0$. Thus,

$$\begin{aligned} counter_{k+1} &= counter_k \cdot d + depth(h_{k+1}) - limit_w \\ &\geq counter_k + depth(h_{k+1}) - limit_w \\ &\geq \sum_{j=1}^{k+1}(depth(h_j) - limit_w) + counter_0. \end{aligned}$$

Hence, the lemma is valid.

Because the value of $counter_0$ is assigned during the last splayed access, we know that $counter_0 \geq -limit_w$. We also know that $counter_c \leq 0$ and, thus, by Lemma 1 we have that

$$\begin{aligned} &\sum_{i=1}^{c} depth(h_i) - limit_w \cdot (c+1) \\ &\leq \sum_{i=1}^{c} depth(h_i) - limit_w \cdot c + counter_0 \leq counter_c \leq 0 \\ &\Leftrightarrow \frac{1}{c}\sum_{i=1}^{c} depth(h_i) \leq \left(1 + \frac{1}{c}\right) limit_w \leq 2\, limit_w. \end{aligned}$$

The bound is strict: it happens, e.g., when $counter_0 = -limit_w$, $depth(h_1) = (1+d)limit_w$, and $c = 1$.

The above bound for average depth holds for all of $H_c \subseteq H_n$. Thus writing $|H_n| = m_n$ and $|H_s| = m_s$, we have the following theorem:

Theorem 2. *Let* $C = n \log_2 n$ *and* $m_n + m_s = m$. *The total access time incurred by* WSPLAY *is at most* $2m_n limit_w + m_s(3 \log_2 n + 1) + C$.

5 Test Setting

As reference algorithms in our empirical evaluation we use splay trees, RBT, and a randomized version of splay trees. All splaying is implemented in a simple top-down fashion. The programming environment is Microsoft Visual C++ 2005 and we use full optimization for speed. The test environment is a PC with a 3.00 GHz Intel Pentium 4 CPU and 1 GB RAM. The cache sizes are 16 kB for L1 and 2 MB for L2.

5.1 The Evaluated Version of Wsplay

The problem in evaluating WSPLAY is to control its two parameters d and $limit_w$. Examining all value combinations would be a massive task. However, moderate changes in the value of parameter d do not affect the results much in practice. Hence, we use a constant value $d = 0.9$.

The parameter $limit_w$ is more problematic because with different values the results vary. However, we are mostly interested in how efficient is splay reduction by monitoring. Thus, a very natural way is to compare its efficiency against data structures that reduce splaying without such monitoring. Randomized splay trees [3,4] do it at random. The data structure of Albers and Karpinski [4], referred here as RSPLAY, matches WSPLAY perfectly in the sense that it contains no other modifications to basic splay trees than reduced splaying.

RSPLAY needs as a parameter the probability p. With the probability $1 - p$ standard BST access is executed instead of splaying. We modify WSPLAY to execute the same amount of splaying by including simple adaptation for the variable $limit_w$. We adjust it in the beginning of every access by adding a value proportional to difference between p and amount of executed splay operations. To prevent algorithm from executing all the available splaying in the beginning, we start the sequence with a $1/p$ length margin where no splaying is allowed. Values of $0, 1, 1/2, 1/4, \ldots, 1/512$ for p are examined. This works well in practice: the difference between p and amount of WSPLAY splaying was at most $\min(0.1\%, 0.05p)$. Note that this modification does not affect the bound of Theorem 2. We can easily keep everything in $O(\log_2 n)$ time given a maximum value for $limit_w$. With this modification it is possible to relate RSPLAY and WSPLAY with different values of p.

5.2 Description of the Data

The keys are inserted in trees in random order. For BSTs this usually leads to a well-balanced tree [20]. We examine only accesses with integer keys in nodes.

The locality of reference is often modeled with *Zipf's distribution* (ZD) [21, p. 400]. In it the ith most commonly accessed item is accessed with a probability p_i inversely proportional to i. We want to experiment with different kinds of ratios of locality or *skewness*. Therefore, we use the modification in which we have an additional parameter $\alpha \geq 0$ [21] so that the ith item has access probability

$$p_i = \frac{1}{i^\alpha C},$$

where $C = \sum_{j=1}^{n}(1/j^\alpha)$ and n is the number of nodes in tree. This distribution is uniform when $\alpha = 0$ and the pure ZD when $\alpha = 1$. The parameter α alone does not give very good control over the skewness of the data. Therefore, Bell and Gupta [7] defined another parameter, the *skew factor*:

$$\beta = \sum_{j=1}^{n/100} p_j.$$

Hence, β is the probability of accessing the 1% of items that is most frequently accessed. With $\beta = 0.01$ the distribution, obviously, is uniform.

To achieve dynamic locality we change the access distribution after every $t = 128$ accesses. When the value of t was very low the results were more like with uniform distribution, but otherwise changing t moderately did not affect the results substantially.

We report the evaluation for tree size $n = 2^{17}$ and $m = 2^{20}$ accesses. We also evaluated the tests at least partially for tree sizes ranging up to 2^{25}. The results were similar with different sizes of trees. However, splay tree benefited from larger tree sizes compared to WSPLAY and RSPLAY. For $n = 2^{25}$ splay tree was at least as efficient as the other two with all the skewness values. For the parameter β we used values $0.01, 0.1, 0.2, \ldots, 0.9$. The values of α in the same order are $0, 0.504, 0.664, 0.764, 0.844, 0.914, 0.981, 1.052, 1.135$, and 1.258.

6 Empirical Evaluation

6.1 Average Depth of an Access

The average access depths for the data structures are depicted in Fig. 1. For WSPLAY and RSPLAY the averages over all values of the parameter p are shown. An interesting observation is that the average access depth for RBT is essentially optimal (recall that $n = 2^{17}$). This may be a reason for the practical efficiency of RBT. Our results resemble those of Bell and Gupta [7]: RBT is superior with low skewness values and with high ones splay trees excel. With low values of β there is no real locality in referencing and, thus, distribution sensitivity is of little use. On the other hand, with high values of β it is useful to raise all the items near the root as soon as possible.

Detailed results for WSPLAY and RSPLAY are presented in Fig. 2. Only results for the best values of p are reported. As expected, WSPLAY was better than RSPLAY with all value combinations of β and p. For both algorithms the same

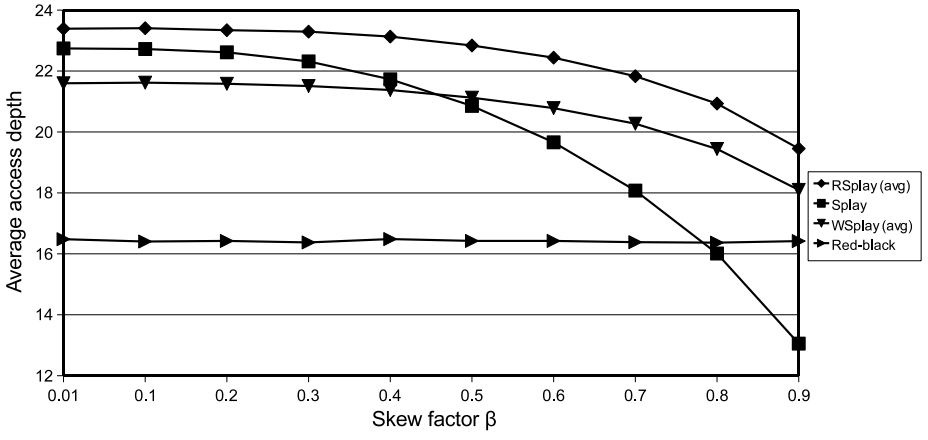

Fig. 1. The average access depths for all data structures

trend for changing the value of p occurs. The best performance for WSPLAY is obtained with $p = 1/16$ and for RSPLAY the best value is $p = 1/4$. The difference between the performance of the algorithms increases with low values of p: With $p = 1/2$ the difference is only 0–2% depending on skewness, with $p = 1/4$ it is already 4–5%, and raises to 7–10% when $p = 1/16$.

The difference of the algorithms on uniform data is probably due to balancing. While RSPLAY does the balances at random, $limit_w$ of WSPLAY settles to a depth in which the amount p of splaying is executed. Thus, the whole tree is treated as a working set. Only roughly the deepest portion p of all accesses are splayed. Splaying the deepest node in a tree tends to halve the depth of a very unbalanced tree [2]. Hence, splaying the deepest nodes in all situations seems to be a better strategy than random splaying. The greater difference in performance between the algorithms for low values of p could also be based on the same reason. For low values of p the only reason to splay is to keep tree balanced and WSPLAY does this better.

As expected, it seems that with high skewness factor values higher values of p were more suitable. It is clearly useful to splay the new working set near the root as soon as possible. Also splaying items near the root does not seem to make the tree more unbalanced. E.g., splaying two different items to the root in turns does not affect the overall balance at all.

6.2 Access Times

The access times for the data structures shown in Fig. 3 are all averages over five evaluations. On the whole the results resemble those of average access depths. RBT is even more superior in these results and splay trees take the most benefit from locality of reference. The overall decrease in access time for splay tree from $\beta = 0.01$ to $\beta = 0.9$ is nearly 50%. An unexpected result is that also RBT decreases its access time by 35% without restructuring the tree.

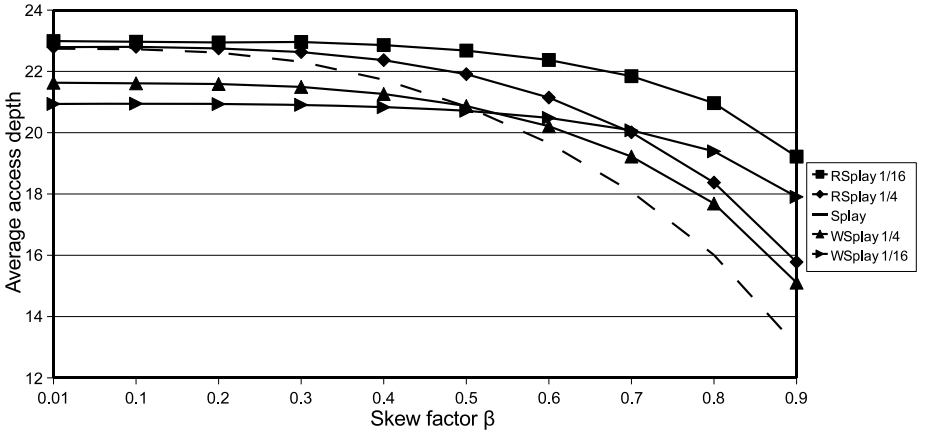

Fig. 2. The average access depths of WSPLAY and RSPLAY with different values of p. The curve for splay tree is marked with the dashed line.

Also the most efficient WSPLAY and RSPLAY are presented in Fig. 3. Understandably the running times decrease as the skewness increases. Also the relative success of the algorithms with different values of p and β is similar to average depths. However, the costly monitoring in WSPLAY reduces its absolute performance. Nevertheless, the most efficient WSPLAY ($p = 1/32$) is usually 5% faster than the most efficient RSPLAY ($p = 1/16$). With $\beta = 0.9$ the difference is only 2%. With low values of β WSPLAY is 7–12% faster than the original splay tree.

With values $p < 1/4$ WSPLAY is 3–5% more efficient than RSPLAY, but with values 0, 1, 1/2, and 1/4 of p the latter prevails. This seems to indicate that in WSPLAY the cost of monitoring the input is compensated only when the amount of splay operations is small. With larger values of p RSPLAY reacts to the change of distribution soon enough. Also RSPLAY seems to need a little more splaying for similar effect, because random splay operations are often not useful. However, absolutely both data structures are most efficient with values $1/4 \geq p \geq 1/64$.

Also with uniform distribution large values of p do not have as good balancing effect as with lower values. In other words, also WSPLAY does too much splaying on accesses that are not very deep in the tree. Thus, restructuring may set the tree out of balance. However with lower values of p WSPLAY is able to splay only the deepest accesses and thus keep the tree balanced.

However, an interesting discovery raises if we examine the efficiency of WSPLAY and RSPLAY for parameter value $p = 1$ (always splay). By comparing these to the original splay tree we get a picture of the implementation specific overhead for the monitoring in WSPLAY and decision-making in RSPLAY. WSPLAY uses 4–7% and RSPLAY 1–3% more time than a splay tree. The access times for RSPLAY do not include the generation of random numbers. This raises the question whether to compare WSPLAY to splay or WSPLAY with $p = 1$.

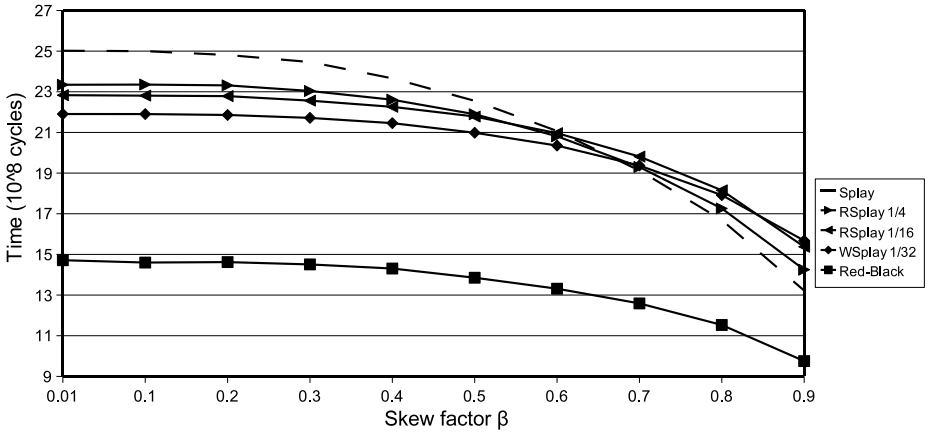

Fig. 3. The average access times for all data structures. For WSPLAY and RSPLAY the best values of p are shown. The curve for splay tree is marked with the dashed line.

WSPLAY increases its efficiency at most more than 15% when compared to the version where WSPLAY always splays. Similar value for RSPLAY is 10%.

6.3 On the Suitability of Zipf's Distribution

Let us briefly analyze the observation that also RBTs seem to be more efficient with higher locality of reference when ZD is used as input (Fig. 3). To the best of our knowledge this has not been reported before.

In Fig. 1 the average depth of RBT does not change. Hence, that is not the reason for the observation. We made sure that setting off compiler optimizations and altering the interval of changing the distribution (value t) did not matter. However, decreasing the size of tree n reduced the advantage of locality of reference down to 25%, but even with very small trees (e.g., $n = 8$) the phenomenon occurred. This contradicts the observation of Bell and Gupta [7] who had a tree of size 4 095. A natural explanation for this is that both operating system and hardware efficiency have progressed significantly since their evaluation. In particular, cache management has progressed in recent years.

In fact, caching could be the reason for this phenomenon. ZD weights the most accessed items very much and thus, with a high probability, only few items are accessed. Let us assume a cache so small that only one path to a node in the whole tree fits it. With high locality even this is useful because a single item is accessed most of the time and, thus, very often the access path is already in cache. Modern memory hierarchy generally consists of many cache layers and for every one of them there is an amount of nodes or paths that fit in. Thus, ZD makes static BSTs use caching very efficiently [22]. Paging complicates the analysis in practice, but the basic idea is the same. A splay tree does not gain as much benefit from caching. In it the access times are more related to the average

depth of an accesses in the tree. A natural cause for this could be the amount of restructuring and memory writing the splay tree does.

We also tried the same evaluation with high skewness but with only one level of working sets. The items in the working set were accessed multiple times uniformly. With a changing probability items were also accessed uniformly outside the working set. The same phenomenon did present itself only slightly. In this case fitting only some of the items in the cache is not at all as useful as it is for ZD. This issue, of course, ought to be studied more thoroughly. Nevertheless, it raises the question of how cautious we should be when generalizing the results with ZD. However, there are studies with real-life data that rank splay tree very efficient in certain situations [9]. Maybe other alternatives (e.g. Lévy distribution [23] or actual web page requests) for modeling locality of reference should be used.

7 Conclusion

This work studied whether it is possible to gain advantage for splay trees by taking the properties of the access sequence into account. On one hand, both the average access depth and time were reduced when compared to randomized splay. This was also the case when compared to splay with high skewness. On the other hand, splay trees excelled with high locality of reference. Also red-black trees were still usually more efficient in these experiments.

Further variations for our algorithms can easily be designed. For example the method can be applied to most of the splay tree versions introduced in [2]. These versions include bottom-up splaying and semi-splaying. Also executing a window of splay operations during change of working set is possible.

Acknowledgments

We would like to thank the anonymous reviewers for insightful and helpful comments. This work was supported by Academy of Finland projects Intents (206280), Alea (210795), and "Machine learning and online data structures" (119699). Moreover, the work of T. Aho and J. Kujala is financially supported by Tampere Graduate School in Information Science and Engineering.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press, Cambridge, MA (2001)
2. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. Journal of the ACM 32(3), 652–686 (1985)
3. Fürer, M.: Randomized splay trees. In: Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, MD, pp. 903–904. SIAM, Philadelphia, PA (1999)

4. Albers, S., Karpinski, M.: Randomized splay trees: Theoretical and experimental results. Information Processing Letters 81(4), 213–221 (2002)
5. Lai, T.W., Wood, D.: Adaptive heuristics for binary search trees and constant linkage cost. In: Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, pp. 72–77. SIAM, Philadelphia (1991)
6. Williams, H.E., Zobel, J., Heinz, S.: Self-adjusting trees in practice for large text collections. Software: Practice and Experience 31(10), 925–939 (2001)
7. Bell, J., Gupta, G.: An evaluation of self-adjusting binary search tree techniques. Software: Practice and Experience 23(4), 369–382 (1993)
8. Heinz, S., Zobel, J.: Performance of data structures for small sets of strings. Australian Computer Science Communications 24(1), 87–94 (2002)
9. Pfaff, B.: Performance analysis of BSTs in system software. ACM SIGMETRICS Performance Evaluation Review 32(1), 410–411 (2004)
10. Lee, E.K., Martel, C.U.: When to use splay trees. Software: Practice and Experience 37(15), 1559–1575 (2007)
11. Denning, P.J.: Working sets past and present. IEEE Transactions on Software Engineering 6(1), 64–84 (1980)
12. Subramanian, A.: An explanation of splaying. Journal of Algorithms 20(3), 512–525 (1996)
13. Badr, G.H., Oommen, B.J.: Self-adjusting of ternary search tries using conditional rotations and randomized heuristics. The Computer Journal 48(2), 200–219 (2005)
14. Iacono, J.: Alternatives to splay trees with $O(\log n)$ worst-case access times. In: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms, Washington, DC, pp. 516–522. SIAM, Philadelphia (2001)
15. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Communications of the ACM 28(2), 202–208 (1985)
16. Amer, A., Oommen, B.J.: Lists on lists: A framework for self-organizing lists in environments with locality of reference. In: Àlvarez, C., Serna, M.J. (eds.) WEA 2006. LNCS, vol. 4007, pp. 109–120. Springer, Heidelberg (2006)
17. Dhodapkar, A.S., Smith, J.E.: Managing multi-configuration hardware via dynamic working set analysis. In: Proceedings of the 29th Annual International Symposium on Computer Architecture, Anchorage, AK, pp. 233–244. IEEE Computer Society, Los Alamitos (2002)
18. Seidel, R., Aragon, C.: Randomized search trees. Algorithmica 16(4), 464–497 (1996)
19. Cheetham, R.P., Oommen, B.J., Ng, D.T.: Adaptive structuring of binary search trees using conditional rotations. IEEE Transactions on Knowledge and Data Engineering 5(4), 695–704 (1993)
20. Martínez, C., Roura, S.: Randomized binary search trees. Journal of the ACM 45(2), 288–323 (1998)
21. Knuth, D.E.: The Art of Computer Programming. Sorting and Searching, 2nd edn., vol. 3. Addison-Wesley, Boston (1998)
22. Brodal, G.S., Fagerberg, R., Jacob, R.: Cache oblivious search trees via binary trees of small height. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, pp. 39–48. SIAM, Philadelphia, PA (2002)
23. Applebaum, D.: Lévy processes — from probability to finance and quantum groups. Notices of the American Mathematical Society 51(11), 1336–1347 (2004)