
When to use splay trees



Eric K. Lee^{*,†} and Charles U. Martel

Department of Computer Science, University of California at Davis, Davis, CA 95618, U.S.A.

SUMMARY

In this paper we present new empirical results for splay trees. These results provide a better understanding of how cache performance affects query execution time. Our results show that splay trees can have faster lookup times compared with randomly built binary search trees (BST) under certain settings. In contrast, previous experiments have shown that because of the instruction overhead involved in splaying, splay trees are less efficient in answering queries than randomly built BSTs—even when the data sets are heavily skewed (a favorable setting for splay trees). We show that at large tree sizes the difference in cache performance between the two types of trees is significant. This difference means that splay trees are faster than BSTs for this setting—despite still having a higher instruction count. Based on these results we offer guidelines in terms of tree size, access pattern, and cache size as to when splay trees will likely be more efficient. We also present a new splaying heuristic aimed at reducing instruction count and show that it can improve on standard splaying by 10–27%. Copyright © 2007 John Wiley & Sons, Ltd.

Received 8 June 2005; Revised 20 November 2006; Accepted 22 November 2006

KEY WORDS: splay tree; cache performance; empirical; heuristic

1. INTRODUCTION

A critical task in designing an application is selecting the correct data structures. This requires an understanding of which data structures will yield the best performance. An asymptotic analysis of the candidate data structures suggests which one will perform best as the input gets large. However, for data structures with similar asymptotic complexity, it is often difficult to predict which will actually perform better in practice.

A further complication is that the reliability of predicting performance based on an analysis under the random-access-machine (RAM) model has lessened in the past 20 years owing to changes in computer architectures. In particular, there now exists an extremely large difference between the performance of the central processor and main memory—some access times for data are hundreds of times longer

*Correspondence to: Eric K. Lee, Department of Computer Science, University of California at Davis, Davis, CA 95618, U.S.A.

†E-mail: leeer@cs.ucdavis.edu

than others. How a data structure minimizes this difference by utilizing the hardware cache can have a significant impact on its execution time.

The binary search tree (BST), and its self-adjusting cousin, the splay tree[‡], are two data structures where the cache plays an important role in performance, so asymptotic analysis is not always a good predictor for these data structures. A randomly built BST is often a good choice for an application that requires fast insertion, deletion, and lookup of items, as well as fast ‘ordered’ operations such as traversals or range queries[§]. However, if the lookup distribution is heavily skewed, a splay tree might be a better choice. In heavily skewed distributions there is a small set of items that are accessed frequently, forming a ‘working set’. Sleator and Tarjan [1] have proved that splay tree access times to the items in a working set are fast. This is true even if the working set changes over time.

Experimental results by Bell and Gupta [2], and more recently by Williams *et al.* [3], have shown that even when the access pattern is heavily skewed, the instruction overhead involved with the splay operation outweighs the instruction savings of having shorter access paths. Bell and Gupta [2] showed that a randomly built BST answered queries faster than a splay, even though a splay tree performs best in a skewed setting. In contrast with these previous results, our results offer a more complete picture of splay tree performance. In particular, we found that owing to superior cache performance there are settings when a splay is faster than the BST, despite a higher instruction count.

Our major contributions in this paper are new empirical results regarding the instruction count, cache performance, and execution time of BSTs and splay trees under skewed lookup patterns. In particular, we show that splay trees have fewer cache misses compared with BSTs, and that the difference in their cache performance increases as the tree grows. The improved cache performance of the splay tree at large tree sizes more than compensates for the higher instruction cost involved in splaying, making them faster than BSTs in this setting. Furthermore, as the cost of a cache miss increases in the future, we expect that the speedup due to splaying will increase. Based on these results we offer guidelines in terms of tree size, access pattern, and cache size as to when a splay tree will likely be more efficient than its static counterpart (BST). We also present a new splaying heuristic aimed at reducing instruction count and show that it can improve upon standard splaying by 10–27%.

Since splay trees are potentially better than BSTs only when the access pattern is skewed, we focus on this setting in this paper. Intuitively, splaying when the access pattern is near uniform is not productive. In a uniform setting there is no payoff for the investment of bringing items to the root since the probability that the same item will be accessed again soon is small. We work under the assumption that a splay tree is desirable because either the exact access distribution is unknown *a priori*, or that the set of frequently accessed items changes over time. If the access pattern is known and fixed, it is trivial to build a tree that is optimal (or near optimal) for the access pattern. Finally, we note that there are many settings when access patterns are heavily skewed but the specific access probabilities are unknown or are dynamic. A simple example is storing movie information in a tree, keyed by title. Certain movies are much more popular than others, but the specific set of popular movies may change over time.

[‡]A splay tree moves accessed items to the root, see Section 2.

[§]If ‘ordered’ operations are not necessary, hash tables are probably a better alternative.

2. BACKGROUND

2.1. Splaying

A splay tree is a self-adjusting BST that improves on the asymptotic bounds in the *amortized* sense, and requires no additional space (in contrast to, say, an Adelson-Velsky–Landis (AVL) tree). Splaying was developed by Sleator and Tarjan [1], and consists of a sequence of operations that moves the queried node to the root of the tree. By performing this move-to-root operation in a particular manner, any sequence of $m > n$ lookups on a n node tree takes only $O(m \log n)$ time. This is an improvement over the $O(mn)$ worst case of a standard BST. Furthermore, when the access pattern is skewed, the frequently accessed nodes will cluster near the root making future access to them faster.

Two main methods for splaying exist, top-down and bottom-up. In our experiments [4], we have found that *simple* top-down splaying, a top-down variant, performs best. Thus, all the results we present involve this variant. In contrast with bottom-up splaying, in top-down splaying only a single pass is necessary to find the item of interest and move it to the root. Details of this method are provided in Appendix A.

2.2. Faster splay trees via heuristics

There are a number of existing heuristics aimed toward improving the performance of splay trees. Most focus on decreasing the overhead involved in splaying by reducing rotations. Williams *et al.* [3] discuss (and evaluate) several heuristics in the context of large text collections [3] (see also experimental results from Bolster [5]). One particular heuristic, randomized splaying, has been shown to perform well in practice. The heuristic was proposed by Albers and Karpinski [6] (a similar method was suggested by Furer [7]). Randomized splaying is based on a simple concept: with probability p a node is splayed, and with probability $1 - p$ a standard BST lookup is performed (and the node keeps its place in the tree).

We consider a deterministic version of randomized splaying—splaying every k lookups and performing a standard BST lookup otherwise. The deterministic variant removes potentially expensive random number generation, which improves performance in practice [6]. We note that the choice of k is somewhat arbitrary, but Albers and Karpinski suggest that $k = 2$ (i.e. alternating splaying and standard BST searching) should provide an improvement over standard splaying in almost all cases. We will refer to this deterministic version of randomized splaying as k -splaying.

We propose a heuristic based on a ‘sliding window’. A sliding window is a queue of fixed size used to keep track of information regarding the last w events. At a high level, our heuristic works as follows. Assume we divide a splay tree creating ‘upper’ and ‘lower’ portions. Now, if most lookups are to the upper portion of the tree then we assume that the frequently accessed items are near the root, so no splaying is necessary. On the other hand, if there are many accesses to the lower portion, splaying is performed under the assumption that the tree could be readjusted to perform better. The sliding window is used to record the upper and lower accesses for the past w accesses.

2.3. Caches

As the disparity between processor and memory speed continues to grow, computer architects have introduced new levels into the memory hierarchy to help bridge the gap. The memory hierarchy exploits the principle of locality—a recently accessed item is likely to be accessed again in the near future.

Therefore it is desirable to keep such an item at a place where it can be quickly accessed. The first level of the hierarchy is closest to the processor and has the fastest access time, but also has the smallest capacity. Since there is a trade-off between the capacity and access time, each successive level is larger but slower. The level(s) between the main memory and the processor are known as *caches*. Typically, the cache closest to the processor is referred to as the L1 cache and for each successively cache $n = \{2, 3, \dots\}$ is the L_n cache. In our results we quantify a data structure's cache performance in terms of cache hits and misses.

Assuming that we have only a single cache between the CPU and main memory, cache hits and misses occur as follows. When the CPU makes a request for a data item the cache is checked; if that item is in the cache the result is a *cache hit*. If the requested item is not in the cache then the item is copied from its location in main memory into the appropriate cache set. This is known as a *cache miss*. The associated cost of bringing the item in from memory is known as the *memory latency* and can be 100 times more expensive than a cache hit [8]. The process of checking successively lower levels of the memory hierarchy has a straightforward generalization to handle cases when there are many cache levels. In our analysis we will focus on capacity and conflict misses. Capacity misses occur when the cache is too small to contain all the blocks needed during the execution of a program. In contrast, conflict misses occur when two or more data items in the memory map to the same cache set. This can occur regardless of the cache size. See Hennessy and Patterson [9] for further details regarding these types of cache misses.

2.4. Related work

Shortly after their creation, splay trees were shown to perform better than some standard heaps in a priority queue study by Jones [10]. This is the result of the fact that no comparisons are needed to find the minimum in a tree (simply follow the left branch) and that the self-adjusting behavior of the tree allowed for shorter paths to the items. Furthermore, in Jones's implementation splay rotations were reduced and specialized for enqueue and dequeue.

Bell and Gupta [2] performed an empirical study where the central focus was a comparison of the performance of splay trees to randomly built BSTs and AVL trees. Trees consisting of 4095 integers were tested with Zipf-like access patterns. A representative result of their findings concerns the case where 100 000 total operations were performed—20 000 updates and 80 000 searches. In this setting the randomly built BST outperformed the splay tree in terms of execution time for all cases except at the highest skew level, which had 90% of all accesses going to 1% of the nodes. In this extremely skewed distribution, splaying had an execution time 6% faster than the BST. The major difference between Bell and Gupta's results and our own is the execution time influence of the cache. In particular, the trees in their experiments were small enough to fit entirely into the cache, thus cache effects had little performance impact.

More recently, Williams *et al.* [3] studied the performance of self-adjusting data structures on the specific task of storing words found in a large text collection. The experiment selected each word in sequence from a 1 GB text file and then performed a lookup for the word in the tree. If the word existed in the tree then some statistical information was updated, and if it did not a new node was created to store the word. The text file was Zipf distributed (see Section 3)—it contained 2.0 million distinct words among 171.4 million total words.

Williams *et al.* [3] reported that BSTs were faster than the various splaying techniques in their experiment, despite the fact that they had a large tree size. However, these results are not contrary to

ours because their experimental setting was significantly different. The primary difference is that our trees were randomly built and their trees were not. Williams *et al.* noted that many of the most common words were likely to appear early in the text collection. Since the words were processed from the start of the collection, the resulting BST has short access paths to these common words. Thus, readjusting a tree constructed in this manner via splaying is likely to offer little benefit. We note that, in a setting like this, sampling the text to find the most common words and inserting these first may be a better alternative to using splay trees.

Iancu *et al.* [11] evaluated splay trees and B-trees with respect to cache performance. Their results show that B-trees were faster than splay trees in answering queries for a variety of access patterns. The exception was for a heavily skewed pattern with high locality, in which the splay was faster. Our results differ because our measurements allow us to determine how much each factor contributes to the performance of the trees. In particular, for the skewed access pattern tested by Iancu *et al.*, the splay outperformed the B-tree, but in addition to having better cache performance the splay also executed only one-third of the B-tree's instructions. Thus, one cannot conclude how much of an effect the superior cache performance of the splay had on the execution time. Furthermore, our results allow us to offer recommendations in terms of tree size and cache size as to when the improved cache performance of the splay outweighs its instruction count overhead relative to BSTs.

3. COMPARISON METHODOLOGY

3.1. Access patterns and tree sizes

Non-uniform distributions are all around us; for example, the distribution of people in U.S. cities, or the distribution of grades for an exam. We focus on one common skewed distribution, Zipf's, named after Harvard linguist George Zipf [12]. Furthermore, this is the distribution previous studies have used to compare splay trees with other data structures [2,3]. We will now briefly describe Zipf's distribution.

Consider a distribution over a set of n objects where the i th most frequently occurring object will appear $1/i$ times the frequency of the most frequently occurring object. In Zipf's distribution the most frequently occurring object appears with probability

$$\frac{1}{\sum_{i=1}^n 1/i} = \frac{1}{H_n}$$

and thus the probability of item i occurring is

$$p_i = \left(\frac{1}{i}\right) \cdot \frac{1}{H_n}$$

where H_n is the n th harmonic number. In essence, Zipf's distribution means that a small portion of the items (e.g. 10%) occur extremely frequently (e.g. 90%). Zipf's law applies to a wide range of data sets. Two examples are the word frequencies in the English language [12], and the frequency of Web page accesses[¶] [13].

[¶]Breslau *et al.* [13] state 'the distribution of page requests generally follows a Zipf-like distribution where the relative probability of a request for the i th most popular page is proportional to $1/i^\alpha$, with α typically taking on some value less than unity'.

We present results involving two types of access patterns, each consisting of 1 million Zipf distributed queries. The *static Zipf* access pattern was constructed as follows.

- (1) For all n keys, select key i uniformly at random among the keys that have not been assigned an access probability. Assign it the access probability $p_i = (1/i) \cdot 1/H_n$.
- (2) For all 1 million queries the probability query j is to key i is p_i .

The second type of access pattern, *dynamic Zipf*, is constructed similarly except that it reassigns probabilities periodically. The difference from the static case is that every 50 000^{||} queries the access probabilities to the tree keys were reassigned using the above process.

Our experiments involved trees ranging in size from 16 384 (2^{14}) nodes up to 1 048 576 (2^{20}) nodes in powers of 2. These sizes were chosen so that the smaller splay trees would fit entirely in the cache, and the larger splay trees would only partially reside in the cache. For instance, the largest tree tested was 24 times larger than the L2 cache.

3.2. Implementation

All implementations were written in C, and compiled using GCC 3.2.2 without optimization. The splay results we present are for the simple top-down splaying variant (see Appendix A for details) since it was the fastest among the variants we tested [4]. The implementation was taken from Weiss's data structures text [14]. The BST implementation was also from Weiss [14]; however, the non-recursive versions of the relevant operations (`insert` and `find`) were used.

The driver program first constructed a binary tree containing n unique integer keys inserted uniformly at random**. We used the same construction routine for building both the splay and BST. This meant that no splaying was performed during the construction of the splay tree. The benefit of constructing the trees in this manner was that the execution times of the two data structures were identical modulo the time to answer queries. Furthermore, splaying during construction is unproductive since the keys were inserted in random order. We note that using the same construction routine for both trees is consistent with the approach used by Bell and Gupta [2].

After the tree was built, the entire query set (static or dynamic Zipf) was read into a large queue (an array of 1 million cells). This step aimed to reduce file I/O during the experiment, which would likely skew the timing results. Despite the queue's size, both the queue and the tree fit entirely in main memory. Furthermore, the queue had a negligible effect on the cache performance of the tree because each cell was only accessed once, and only in a localized fashion. Thus, the queue only utilized a small portion of the cache at any given time during the experiment. Finally, once the tree was constructed, and the queries were stored in memory, the driver processed each query using the tree's search routine.

^{||}This quantity was chosen because in limited experimentation we determined that 50 000 was approximately the limit where splay trees begin to lose their effectiveness. Decreasing the quantity further meant the splay trees would perform worse than BSTs for all tree sizes.

**Timing skew introduced by random number generation was avoided by producing a file offline that contained the integers to be inserted.

3.3. Test system

All timing measurements were performed on a 1 GHz Pentium III machine, running Red Hat Linux 9.0. The test system had 512 MB of main memory, and a 512 KB eight-way set associative L2 cache. In addition, it had two L1 caches (instruction and data) both 16 KB in size with eight-way associativity. The latency of main memory was measured (using RightMark memory analyzer [15]) to be approximately 100 ns, and 8 ns for the L2 cache. The L1 caches had a 2 ns latency on a cache hit, but as a result of pipelining the L1 cache could be accessed without delay^{††}.

Throughout our analysis we will assume that the test system executes 870 million instructions per second. The SPEC integer benchmarks in Hennessy and Patterson's text [8] provide an average cycle per instruction (CPI) of 1.15 for the P6 micro-architecture (base architecture for the Pentium pro, Pentium II and Pentium III). Using this CPI and a clock rate of 1 GHz gives an execution rate of approximately 870 million instructions per second.

3.4. Trials and metrics

Cache and instruction counts were performed using the `sim-cache` simulator, one of the tools that make up the SimpleScalar tool set [17]. SimpleScalar is an execution-driven simulator that executes a compiled version of the target program. This allows for a very detailed analysis of the candidate program since every machine level instruction and data access can be tracked. Another benefit is that the candidate program need not be instrumented with profiling code that could bias the results.

We modified the simulator to measure only the cache misses and instruction counts of the trees during the queries (the only part of the simulation where their performance could differ). More accurate simulation results were obtained by first performing 500 000 'warm up' queries followed by 1 million queries during which the performance was measured. The warm up phase allowed us to measure the steady state performance of the trees.

In addition to simulated results, real time measurements were collected. The timing experiments were run immediately following system boot, with the system running a light load (CPU usage was consistently between 98 and 100% over the lifetime of a run). The time was measured using the Linux `time` utility, which gives the end-to-end execution time of the program. We collected data for three trials of identical query sequences for every tree size and access pattern combination. The minimum time among the three trials is what we report, as this time reflects the least CPU and cache interference from other OS functions. However, the variance among the three trials was negligible. Experiments using different random number seeds were also tested, but there was no significant variance among those either.

The times we report include the time to read in the query file, build the tree, and perform the queries. The following procedure was used to obtain sufficiently long execution times and assure the majority of time was due to the queries: a total of 10 million queries were performed by repeating the 1 million queries in the query file from start to end 10 times. By reusing the same set of queries we were better able to note the differences in the execution time in two ways. First, the longer execution times (typically greater than 7 s) reduced the variance, and produced more accurate results.

^{††}The load unit on the Pentium III has a throughput of 1, as fast as any other functional unit in the pipeline [16].

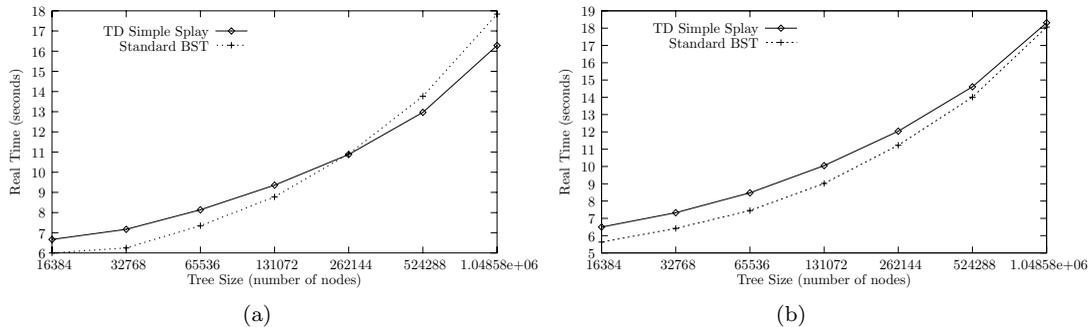


Figure 1. Times include reading the data file, building the tree, and performing the queries: (a) the static timing results for 10 million queries with the access pattern held constant; (b) the dynamic timing results when the lookup probabilities were reassigned every 50 000 queries.

Second, repeating the same queries made the proportion of time spent on reading in and storing the queries smaller. For all tree size and access pattern combinations, more than 75% of the total end-to-end time was from the lookups.

4. RESULTS

4.1. BST versus splay—instruction counts and execution times

We first tested how much better the structure of a splay tree was in terms of average search depth, when the lookups are Zipf distributed. We found that for the static Zipf access pattern the splay tree visited approximately 40% fewer nodes per search compared with the BST. In particular, for the largest tree size (2^{20} nodes) the splay performed 10 fewer compares and 10 fewer pointer assignments per search.

Despite the superior structure of the splay tree, Figures 1(a) and 1(b) show that the execution time of the splay tree is as much as 14% worse than that of the standard BST for both access patterns. Figure 1(a) shows the timing results for the static Zipf distribution. Interestingly, at the largest tree size the splay tree outperforms the standard BST by about 8%. Figure 1(b) gives the timing results for the dynamic Zipf queries. For the dynamic access pattern the splay tree was slower than the BST for all tree sizes. However, as in the static case, the gap decreased as the tree size increased. At the largest tree size the execution times were nearly identical.

One known reason why splay trees are slower than BSTs is that the number of comparisons and pointer assignments during splaying outweigh the savings in search depth [2,3]. To confirm this, we obtained machine level instruction counts using SimpleScalar. Figure 2 shows that the BST performs 16–18% fewer instructions across *all* tree sizes. Furthermore, by projecting using the slopes of the curves from Figure 2 it is evident that, even for a tree with more than 10 million nodes, the splay will still perform more instructions than the BST. Even though the instruction counts are consistently higher for the splay tree, the timing results indicate that in the static case the splay is faster at large tree sizes.

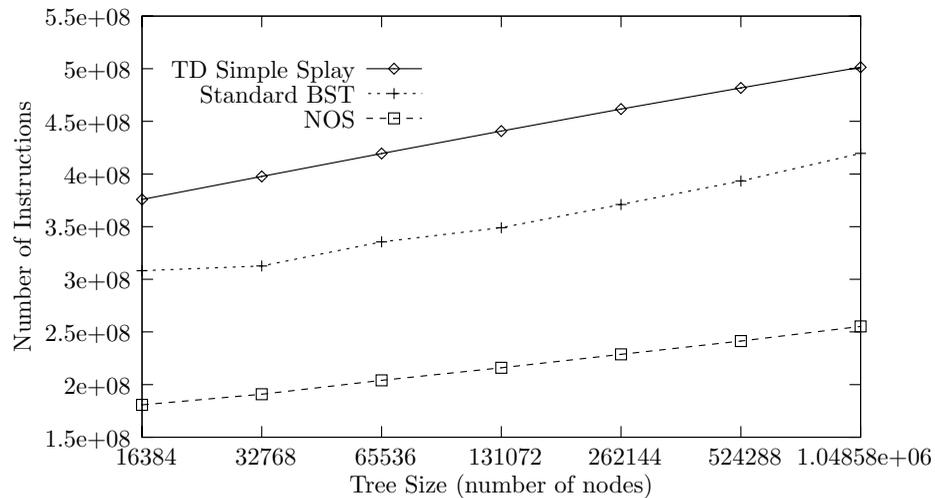


Figure 2. Instruction counts from SimpleScalar for the splay and BST on the static Zipf access pattern. The instruction counts were for the 1 million queries (excluding construction), measured after 500 000 ‘warm up’ queries.

Thus, instruction overhead cannot completely account for the difference in execution time of the two trees.

In addition to the instruction counts for the BST and splay, we also tested a third data structure: the ‘near optimal splay’ (NOS) tree. The NOS was designed to provide a hypothetical upper bound to splay tree performance. In particular, if there was no overhead in splaying, the splay tree would perform similarly to the NOS tree. The NOS performs lookups exactly as a standard BST. However, prior to the lookups, a one-time restructuring is performed so that the most frequently accessed nodes are at the top of the tree. This is accomplished by splaying the most frequently accessed nodes to the root in *reverse* order of their access probabilities. Thus, for the static Zipf access pattern the NOS keeps the most frequently accessed items close to the root without further splaying.

The instruction counts for the NOS tree on the static Zipf access pattern in Figure 2 are particularly interesting. In terms of absolute savings, it performs about 1.6×10^8 fewer instructions than the standard BST for the largest tree size. By using the assumption of 870 million instructions per second for our test machine, a saving of 1.6×10^8 instructions over a million queries translates to about 1.83 s over 10 million queries. Examining the differences in execution times for the largest tree size in Figure 1(a) reveals that the standard splay tree has about a 1.8 s speedup over the BST. That is, although the NOS executes significantly less instructions than the BST, and the standard splay tree significantly more, they both exhibit a similar speedup advantage over the BST. This implies that the splay tree’s execution time is determined by other factors besides instruction counts.

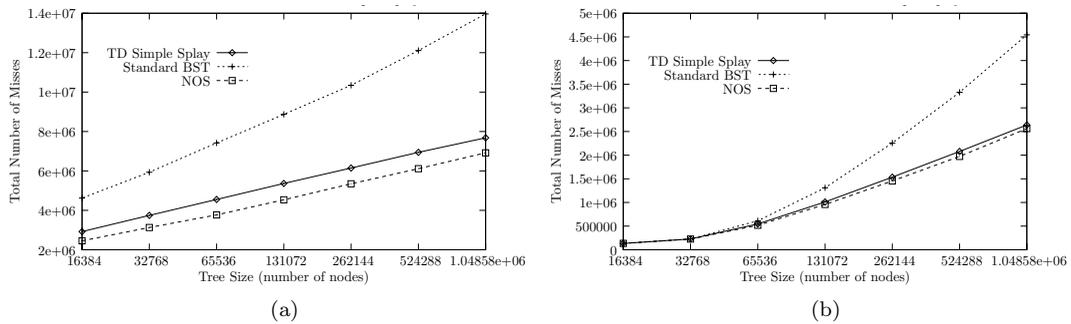


Figure 3. (a) L1 and (b) L2 cache performance of splay and BST on the static Zipf data set. The miss counts were for 1 million queries (excluding construction), measured after 500 000 ‘warm up’ queries. The simulation parameters matched that of the test system.

4.2. BST versus splay—cache performance

In our experiments the BSTs and splay trees were constructed in an identical manner. Thus the relative locations of the nodes in memory were the same, and furthermore, they had an identical mapping to the cache. Since conflict misses are dependent on the mapping of the tree nodes to cache blocks, these types of misses should be similar for both trees. Thus, conflict misses could not result in a significant difference in cache performance. However, by examining the difference in the locations of the frequently accessed nodes in the two trees as the queries progress, we can conclude that a splay will have fewer *capacity* misses.

For both the BST and splay, the highly accessed nodes are initially randomly dispersed throughout the tree. Since the bulk of the nodes are in the bottom few levels of the trees immediately following construction, most of the frequently accessed nodes are far from the root. For the static BST these nodes never move, so each such frequently accessed node creates a long ‘chain’ of additional frequently accessed nodes from the root to itself (the access probability of an internal node is the sum of the access frequencies of its children). In contrast, the splay tree brings the frequently accessed items to the root of the tree. Thus, the chains are shorter in the splay. From these observations we can conclude that the BST accesses more distinct nodes than the splay tree. Given that the cache is limited in size, the likely consequence of accessing more distinct nodes is more capacity misses.

To test the hypothesis that splay trees have better cache performance than BSTs, we measured the number of L1 and L2 cache misses for the two trees using SimpleScalar. The cache parameters used in these two simulations are identical to the test system used in the timing experiments. The results of these simulations for the static Zipf access pattern are shown in Figures 3(a) and 3(b).

Figure 3(a) shows that as tree size increases, the splay tree has significantly fewer L1 cache misses than the BST. At the smallest tree size the splay has 37% fewer L1 misses than the BST, and at the largest tree size the difference increases to 45%. Figure 3(b) shows a similar trend for the L2 cache. At the three smallest tree sizes there is no significant difference between the two trees,

but at 131 072 nodes the splay performs 23% fewer L2 cache misses. As with the L1 cache this difference increases, and at the largest tree size the splay tree has approximately 42% fewer L2 cache misses.

Using the latency values of our test machine, and the number of cache misses given in Figures 3(a) and 3(b), we can approximate the savings in execution time for the splay tree due to improved cache performance. For instance, for the largest tree the splay has approximately 6.4 million fewer L1 cache misses. At an 8 ns penalty per miss, this results in approximately a 0.05 s saving. For the L2 cache, the splay has approximately 1.9 million fewer misses than the BST, which translates to approximately a 0.19 s advantage. Summing these two times gives a 0.24 s total cache improvement for the splay at the largest tree size. Since the measured miss totals were for 1 million queries, for over 10 million queries (the number of queries used in the timing measurements) we expect around 10 times the difference, or 2.4 s.

With the cache performance speedup in hand, we can approximate the *net* speedup for the splay tree by including the instruction counts given in Figure 2. Given that the splay tree performs about 8.2×10^7 more instructions than the BST at the largest tree size, and using the assumed execution rate of the test system, we have approximately 0.09 s worth of additional instructions performed by the splay tree. Scaling this difference up to 10 million queries gives a 0.9 s difference. Thus, the projected net speedup by the splay tree is $2.4 - 0.9 = 1.6$ s. We can check this calculation with the measured time difference between the splay and BST for the largest tree in Figure 1(b). This measured difference is 1.56 s; thus, this breakdown of cache/instruction contribution is probably accurate.

Recall from Figure 1(b) that, for the dynamic case, the splay was not faster than the BST at any tree size. Again, cache performance is a major reason for these results. Our simulations showed that the splay tree had a much smaller advantage in cache performance in the dynamic setting [4]. Since the frequently accessed nodes are constantly changing, the splay tree visits more distinct nodes than in the static case. The result is similar to the BST—not all of these nodes can be kept in cache at the same time.

In addition to a smaller benefit from the cache, the splay also suffered from an even larger instruction overhead in the dynamic case [4]. Since the frequently accessed nodes are changing, the splay tree never stays in a steady state for very long (if at all). Thus, longer paths are splayed on each lookup compared with the static case. This translates to more pointer assignments and a higher overall instruction count. In contrast, for the BST, the instruction counts are similar for both the static and dynamic lookups. For the dynamic access pattern, each reassignment of lookup probabilities is random. This implies that the average search path length, which determines the instruction count, has little variance.

4.3. Evaluating the sliding window heuristic

The previous section shows that moving the frequently accessed nodes to the root results in better cache performance. Unfortunately, a splay tree performs a large number of instructions to accomplish this. We can reduce this instruction overhead by noticing that once the frequently accessed items are near the top of the tree, further splaying is largely unproductive—it is a waste to bring a frequently accessed node to the root if it replaces another frequent item. Figure 4 illustrates the results of the sliding window heuristic (SW-splay) based on this observation. The results are for the dynamic Zipf access pattern. Notice that every 50 000 queries, the SW-splay detects the changes in the access pattern and adjusts the splaying activity accordingly. Immediately following a change in the access pattern

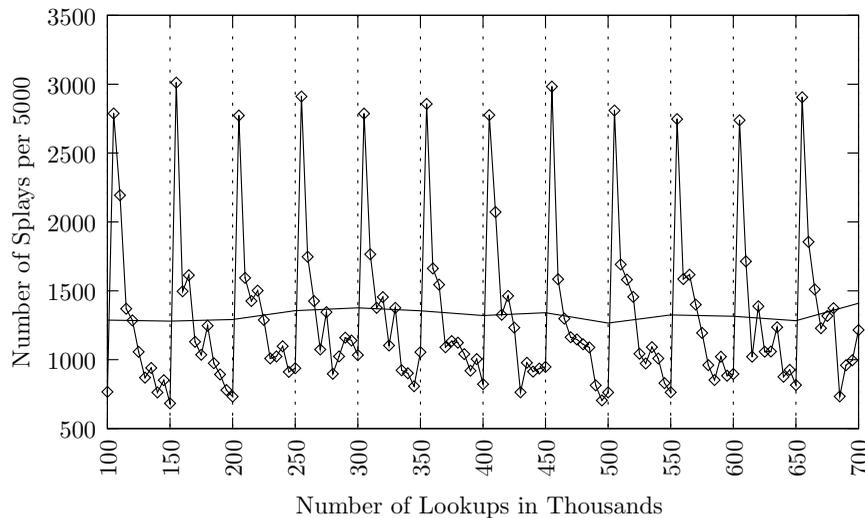


Figure 4. Number of splay operations per 5000 lookups on a 2^{20} node SW-splay tree. The lookups were from the dynamic Zipf access pattern, vertical dotted lines show the changing points. The depth d was 20, and the threshold t was 16, with $w = 32$. The solid line (nearly constant at 1250 splays per 5000) represents the average number of splays per 5000 lookups in a 50 000 query span.

the splaying activity peaks. However, once these nodes are splayed to the root, the frequency of splaying quickly falls off.

More specifically, a *sliding window splay tree* (SW-splay) works as follows.

- We first choose two parameters, d and t . The parameter d is the *depth* of the tree used to differentiate an upper and lower access. For every lookup that traverses at most d nodes, the lookup is counted as an upper access. Otherwise, the lookup is a lower access. The parameter t is the *threshold* determining the required number of upper accesses in the sliding window. That is, if the number of upper accesses in the past w lookups is less than t , splaying is performed on the next lookup. Otherwise, the next lookup will be a static BST search (no rotations).
- On each lookup update the sliding window by dequeuing the result (upper or lower) of the oldest lookup and enqueueing the result of the current lookup. Since it is only necessary to keep track of upper or lower on each access, the sliding window can be implemented efficiently as a bit queue.

We compared the SW-splay against three other trees: a standard BST, k -splaying for four values of k , and a standard splay tree. For all the SW-splay results the depth cutoff, d , was $\log_2 n$ where n is the number of nodes in the tree. The value of the threshold, t , was 16, half the sliding window size. The parameter d was chosen to be the height of a perfectly balanced tree and t was chosen so that on average at least 50% of the accesses were as fast as in a perfect balanced tree. We note that changing the choice of d within a moderate range (e.g. ± 2) does not greatly alter the performance—some values

Table I. Comparison of execution times (in seconds) for various tree sizes for BST, a standard Splay tree, k -splaying, and the SW-splay when the lookups were Zipf distributed and the lookup probabilities were held constant.

n (nodes)	BST	$k = 64$	$k = 16$	$k = 4$	$k = 2$	Splay	SW, $d = \log n, t = 16$
16 384	5.89	4.48	4.59	5.03	5.45	6.36	4.62
32 768	6.13	5.19	5.25	5.66	6.12	7.11	5.35
65 536	7.22	6.13	6.14	6.48	7.06	8.03	6.28
131 072	8.69	7.49	7.37	7.68	8.24	9.29	7.46
262 144	10.87	9.28	9.01	9.13	9.70	10.78	9.00
524 288	13.73	11.86	11.40	11.32	11.81	12.91	11.18
1 048 576	18.09	15.81	15.18	14.89	15.30	16.31	14.75

Table II. Comparison of execution times (in seconds) for various tree sizes for BST, a standard Splay tree, k -splaying, and the SW-splay when the lookups were Zipf distributed, and the lookup probabilities were changed every 50 000 lookups.

n (nodes)	BST	$k = 64$	$k = 16$	$k = 4$	$k = 2$	Splay	SW, $d = \log n, t = 16$
16 384	5.51	4.83	4.76	5.04	5.52	6.39	4.95
32 768	6.31	5.67	5.62	5.83	6.30	7.24	5.82
65 536	7.33	6.85	6.79	7.04	7.60	8.36	7.09
131 072	8.89	8.57	8.43	8.78	9.04	9.96	8.71
262 144	11.06	10.77	10.55	10.57	11.02	11.93	10.67
524 288	13.85	13.76	13.40	13.19	13.60	14.52	13.27
1 048 576	18.07	18.21	17.70	17.19	17.44	18.35	17.10

worked better at the large trees and others worked better at the smaller trees. Also the value of d is related to the choice of t ; if a smaller d value is used, a smaller t value will work better.

Table I shows the timing results of these methods for the static Zipf access pattern (10 million queries). The first noteworthy trend is that both k -splaying (for all values of k) and the SW-splay performed better than both a standard BST and splay tree across all tree sizes. In fact, at the largest tree sizes, 4-splaying and the SW-splay outperformed the BST by 18%. A second interesting result is that no value of k is optimal across tree sizes. For 64-splaying, the reduction in instruction count greatly aids the smaller trees. However, since cache performance is a critical factor at the larger tree sizes, a welcome trade-off is to perform more splaying, which results in more instructions but better cache performance. Thus, the smaller values of k did better at the larger tree sizes. Finally, although the SW-splay offers substantial improvement over the standard splay tree (27% at the smallest tree size and 10% at the largest tree size), the improvement over 4-splaying is small, just 8% at the smallest tree size and a marginal 1% at the largest tree size.

Table II provides the results for the dynamic Zipf access pattern. The same trends as in the static case are apparent, but of additional interest is that no method of splaying substantially improved on the

standard BST at the larger tree sizes. This indicates that while the heuristics are effective at reducing the instruction count overhead in splaying, none improve on the cache performance of the splay trees. Furthermore, it is likely that if the lookups were even more dynamic, for example changing every 10 000 queries, a standard BST would outperform all the heuristics presented here. This is because there would be small (if any) cache benefit from splaying in a highly dynamic case, and any reduction in instruction count from the SW-splay or k -splaying (with small k) would be minimal because of the constant readjusting.

5. CONCLUSION

In this paper we have compared the performance of standard BSTs to splay trees under skewed access patterns. In particular, we have focused on the cache performance of the two trees. Our results demonstrate that splay trees have superior cache performance over BSTs when the lookups are skewed, and that this improvement can result in better overall performance.

We have also proposed a new heuristic for splaying, the SW-splay, that reduces the instruction overhead involved in constant splaying. Our results indicate that the heuristic can be a significant improvement on standard splaying. However, if the a good choice of k can be easily determined, k -splaying is likely to be preferable because it offers a similar level of improvement and has the added benefit of being extremely simple to implement. The SW-splay may be a good choice when k cannot be determined *a priori*. Since the SW-splay keeps a history of lookup results, it has the ability to adjust its parameters online. We have performed limited experimentation in this paper; additional testing and developing methods for adjusting parameters is an area for future work.

Based on our findings we offer some guidelines as to when splay trees are likely to outperform BSTs. A splay tree is likely to have faster lookups than a randomly built BST when both of the following are true.

- The lookup pattern is highly skewed (e.g. Zipf).
- The set of frequently accessed nodes remains constant for a significant number of queries (in our experiments around 200 000), and the tree is not constructed based on the access pattern (as in the text collection experiment by Williams *et al.* [3]).

Furthermore, at least one of the following must also be true.

- The trees are much too large to fit in the cache—either in terms of number of nodes or node size (e.g. large keys such as strings). (In our experiments for the static Zipf case 10 times the L2 cache size was approximately the point where savings in memory latency outweighed the instruction overhead.)
- The system has high memory latency. The current trends in architecture design indicate that memory latency will continue to increase relative to processor performance [8]. Thus, although in our experiments the splay had only an 8% speedup over the BST in the best case, in the future we expect that this speedup relative to BSTs will increase. However, we note that new architectures with better prefetching support may reduce the actual penalty for cache misses. Additional experiments on newer architectures can help to clarify this.

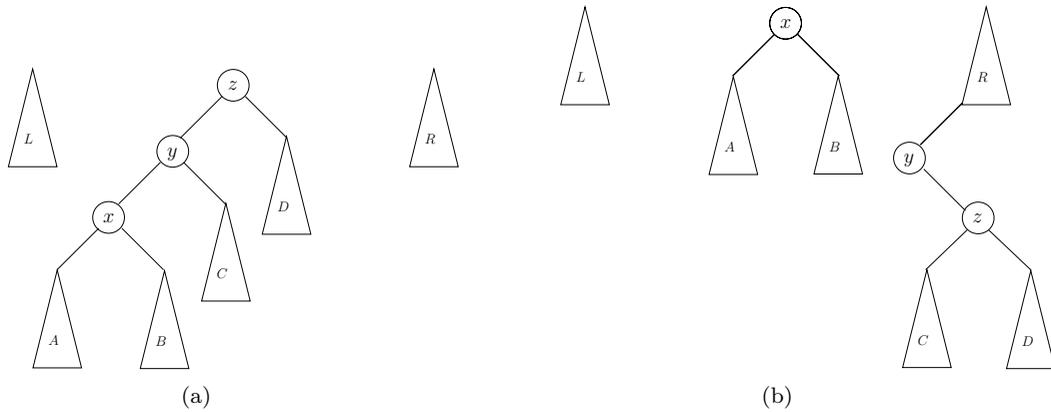


Figure A1. Simple top-down rotation for a zig-zig case: (a) the node x is the item to be promoted; (b) the nodes y and z are rotated and then added to the minimum of the right subtree.

Our experiments involved synthetic data sets with independent though non-uniform accesses. More realistic settings may have a temporal locality of reference or may access keys of similar value more frequently within a time window. Each of these would be favorable for splay trees, but might also improve the cache performance of the BST. Thus, it would be helpful to continue this work on a range of realistic data sets

APPENDIX A. TOP-DOWN SPLAYING

We provide the details of simple top-down splaying. An implementation of this splaying variant can be found in Weiss's text [14]. The tree is first split into three parts: left, right, and middle. In the left subtree are the items known to be smaller than the accessed item, and in the right are the items known to be larger. In the middle are items that are not yet determined to belong to either of the previous trees. Initially the left and right subtrees are empty and the middle tree is the entire tree. The search proceeds two nodes at a time starting with the root node.

If the next two nodes on the search path are both left (right) children (see Figure A1) we have a zig-zig (zag-zag) pattern. If a zig-zig pattern is found, a rotation is done between the parent and grandparent nodes. After performing this rotation, both nodes and their subtrees are added to the minimum of the right tree. If the next two nodes on the path are a left child followed by a right child, we have a zig-zag pattern (see Figure A2). In this case the grandparent is appended to the right (left) tree. The parent node stays in the middle tree. Both the zig-zig and zig-zag have symmetric equivalents and are handled accordingly.

There is the special case when zig-zig and zig-zag rotations do not exactly end at the node searched for, x . In this case the parent of x is added to the minimum of the right tree if it is larger than x , or it is added to the maximum of the left tree if it smaller than x . One final step is performed

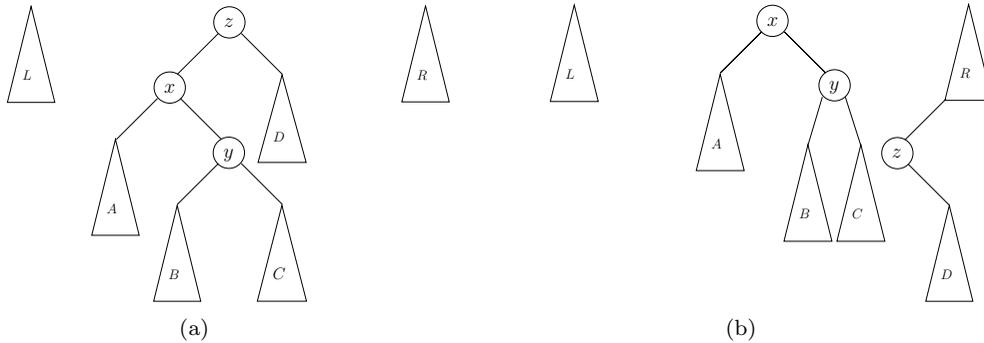


Figure A2. The zig-zag rotation for simple top-down splaying. This case is the only difference from standard top-down splaying: (a) node y is the item to be promoted; (b) only node z is appended to the minimum of the right subtree, and no rotation is performed.

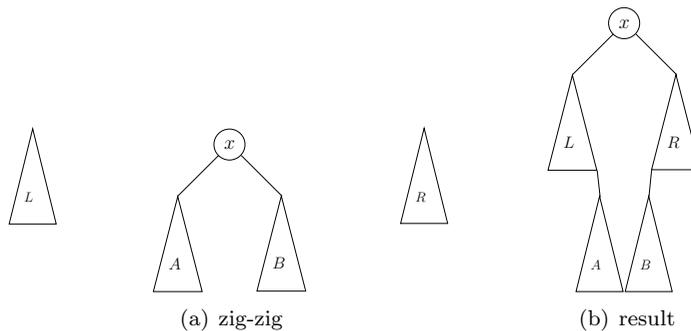


Figure A3. The final step in top-down splaying: (a) zig-zig pattern; (b) the left and right trees are reassembled with x (the splayed item) as the new root.

when reaching x ; the left, right, and middle trees are reassembled. The searched item, x , now becomes the root of the entire tree with the left and right trees becoming the left and right subtrees of the new final tree. Figure A3 shows the reassembling of the three trees.

ACKNOWLEDGEMENTS

We wish to thank the anonymous referee for several helpful comments which improved this paper. We would also like to thank Professor Matt Farrens for useful suggestions and technical advice.

REFERENCES

1. Sleator DD, Tarjan RE. Self-adjusting binary search trees. *Journal of the ACM* 1985; **32**(3):652–686.
2. Bell J, Gupta GK. An evaluation of self-adjusting binary search tree techniques. *Software—Practice and Experience* 1993; **23**(4):369–382. Available at: <http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol23/issue4/spe819.pdf>.
3. Williams HE, Zobel J, Heinz S. Self-adjusting trees in practice for large text collections. *Software—Practice and Experience* 2001; **31**(10):925–939. Available at: <http://www.cs.rmit.edu.au/~jz/fulltext/spe01.pdf>.
4. Lee EK. Cache performance of binary search trees and analyzing victim caches. *M.S. Thesis*, Department of Computer Science, University of California at Davis 2005. Available at: http://www.cs.ucdavis.edu/~leeer/research/ms_thesis.pdf.
5. Bolster A. Experiments in self adjusting data structures. *BIT Honors Thesis*, School of Information Technology, Bond University, 2003. Available at: <http://www.it.bond.edu.au/publications/02THESIS/THESIS02-05.pdf>.
6. Albers S, Karpinski M. Randomized splay trees: Theoretical and experimental results. *Information Processing Letters* 2002; **81**(4):213–221. Available at: <http://citeseer.nj.nec.com/507513.html>.
7. Furer M. Randomized splay trees. *Proceedings of the ACM–SIAM Symposium on Discrete Algorithms*, January 1999. SIAM: Philadelphia, PA, 1999.
8. Hennessy JL, Patterson DA. *Computer Architecture: A Quantitative Approach* (3rd edn). Morgan Kaufmann: San Mateo, CA, 2003.
9. Hennessy JL, Patterson DA. *Computer Organization & Design: The Hardware/Software Interface* (2nd edn). Morgan Kaufmann: San Mateo, CA, 1997.
10. Jones DW. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM* 1986; **29**(4):300–311.
11. Iancu C, Acharya A, Ibel M, Schmitt M. An evaluation of search tree techniques in the presence of caches. *Technical Report UCSB TRCS01-08*, UCSB, 2001. Available at: <http://citeseer.ist.psu.edu/iancu01evaluation.html>.
12. Zipf GK. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Addison-Wesley: Cambridge, MA, 1949.
13. Breslau L, Cao P, Fan L, Phillips G, Shenker S. Web caching and Zipf-like distributions: Evidence and implications. *Proceedings of the 18th Conference on Computer Communications (INFOCOM '99)*. IEEE Computer Society Press: Los Alamitos, CA, 1999; 126–134. Available at: <http://citeseer.ist.psu.edu/559400.html>.
14. Weiss MA. *Data Structures and Algorithm Analysis in C* (2nd edn). Addison-Wesley: Reading, MA, 1997.
15. RightMark. RightMark Memory Analyzer, 2004. <http://cpu.rightmark.org/> [27 January 2007].
16. Intel. Intel Architecture Optimization Reference Manual, 1999. <ftp://download.intel.com/design/PentiumII/manuals/24512701.pdf> [27 January 2007].
17. SimpleScalar LLC. Corporation Website. <http://www.simplescalar.com> [27 January 2007].