Self-adjusting trees in practice for large text collections

Hugh E. Williams Justin Zobel Steffen Heinz
Department of Computer Science, RMIT University
GPO Box 2476V, Melbourne 3001, Australia.
E-mail: {hugh,jz,sheinz}@cs.rmit.edu.au

June 18, 2002

Abstract

Splay and randomised search trees are self-balancing binary tree structures with little or no space overhead compared to a standard binary search tree. Both trees are intended for use in applications where node accesses are skewed, for example in gathering the distinct words in a large text collection for index construction. We investigate the efficiency of these trees for such vocabulary accumulation. Surprisingly, unmodified splaying and randomised search trees are on average around 25% slower than using a standard binary tree. We investigate heuristics to limit splay tree reorganisation costs and show their effectiveness in practice. In particular, a periodic rotation scheme improves the speed of splaying by 27%, while other proposed heuristics are less effective. We also report the performance of efficient bit-wise hashing and red-black trees for comparison.

1 INTRODUCTION

A key task in constructing an index for a large text collection is gathering the distinct words in the collection. In this paper we investigate the use of the splay and randomised search trees—two well-known self-adjusting binary search trees—for managing a large lexicon of words of this kind. Sleator and Tarjan [20], in their original description of the splay tree, propose heuristics for reducing the costs of self-adjustment and note "that the practical efficiency of various splaying and [heuristic] semisplaying methods remains to be determined". One motivation for our work is to augment theoretical analyses [1] by experimenting with efficiency heuristics for index construction in large text databases. In addition, we seek to verify whether it is true that "splay trees are as efficient as balanced trees when total running time is the measure of interest" and whether, as claimed, "they can be much more efficient if the usage pattern is skewed" [20].

Splay and randomised search trees can be used for fast searching and insertion in index construction for large text databases. In this task, a sequence of word occurrences is extracted from the text to be indexed. The number of word occurrences greatly exceeds the number of distinct words, but new words continue to be seen—at a rate of perhaps one in a thousand occurrences [25]—even after many gigabytes of text have been processed. The distribution of words is highly skew: in typical English text around one word in twelve is "the", and around one word occurrence in five is one of "the", "and", "or", "of", or "a". In a large text collection, the majority of distinct words (which are typically typographic errors, nonsense strings, or unusual strings such as chemical names or fragments of genomes) occur once only.

Each word occurrence in the collection is searched for in the tree structure. If the word is new, a node is added to the tree to contain it; otherwise the information about the word is updated (since the purpose of gathering distinct words is to collect statistics about them). In a standard splay tree, a new or accessed node is then splayed to the root of the tree. In a randomised search tree, a node may be rotated towards the root one or more times; this depends on a randomly-generated priority associated with the node, and the relative priority of nodes above the new node.

Index construction might be considered an ideal application of these trees, since the amortised time cost over the data set is important, not the cost of an individual search or insertion. Amortised structures are designed to be usually not much worse than the average performance of a more rigid structure and, in applications where the the access patterns are not uniform, they are potentially more efficient. Indeed, efficient search systems, such as the MG text database [24] and the CAFE indexed genomic search system [22], use splay trees for index construction.

The splay tree is also memory-efficient compared to other structures that might be used for index construction. Other structures—such as the B-tree [3, 20]—have significantly more internal structures to maintain rigid balance. The randomised search tree is less memory efficient as a priority value is stored in each node. However, heuristic approaches that do not store priorities have been proposed.

We report experiments in this paper with unmodified top-down splaying, existing and novel bottom-up heuristic splaying, randomised search trees, and compare these to an unbalanced binary tree, a red-black tree, and hashing as a method for creating and storing large lexicons. (We use BST to denote a standard, unbalanced binary search tree throughout this paper.) We have found that unmodified splaying and randomised search trees are typically slower than using a BST and, moreover, randomised search trees in particular are less space-efficient. These results are despite the strongly skewed distribution of words. Moreover, we have found that heuristics proposed by Sleator and Tarjan [20] typically do not work better than an unmodified splay tree. However, a simple, novel periodic rotation heuristic works well, reducing the vocabulary accumulation time on a large collection by around 27% over an unmodified splay tree. While this heuristic means that the worst case of the modified splay tree is little better than that of a BST, the worst case is much less likely to arise and we do not believe that it would be a significant problem in practice.

We have compared our heuristic splay tree to hashing with a fast, uniform hash function [16]. The hash table is more than 18% faster than the splay tree for inserting data. However, in some cases, trees may be preferred to hash tables for managing collections of words.

2 SPLAY TREES

Splay trees are binary search trees that are incrementally reorganised to avoid the worst-case linear search and insertion time of a BST [20]. The storage costs of a standard splay tree are the same as BSTs; no counters, pointers, or balance information need be stored.

Splay trees do not require that a tree be balanced, but rather that each access reorganises the tree to make the tree, on average, more balanced. This process of reorganisation, or *splaying*, brings each accessed node to the root of the tree, roughly halving the depth of all nodes along the access path. The rationale is that, for data with a skew distribution, some nodes will be accessed more often than others. With incremental reorganisation, such nodes will cluster towards the root of the tree, reducing the traversal costs of searching the tree for common nodes; this is a similar principle to move-to-front reorganisation of linked lists.

Splaying is implemented using *rotations*. The majority of rotations performed are one of two kinds of *double rotation*, a *zig-zag* or *zig-zig*. Rotations are applied to a node until it is the root of the tree. This reorganisation heuristic guarantees for m accesses that the cost of accessing a tree of n nodes is $O(m \log n)$ in time if there are at least as many accesses as there are nodes in the tree, that is, $m \geq n$. However, the access time for a single access in a splay tree may be $O(n)$ in time. Thus splay trees are an amortised structure: the overall cost of accessing nodes is reduced, but not necessarily the cost of accessing an individual node.

Rotation works as follows. A zig-zag rotation is used to move a node two levels towards the root in two cases: when a node is the left child of its parent, and its parent is the right child of its grandparent; and when a node is the right child of its parent, and its parent is the left child of its grandparent. An example of zig-zag rotation is shown in Figure 1, where a node C with a parent P and a grandparent G are zig-zagged.

A zig-zig rotation is used in the other two possible grandparent, parent, and child arrangements: where a node is the left child of its parent, and the parent is the left child of the grandparent;

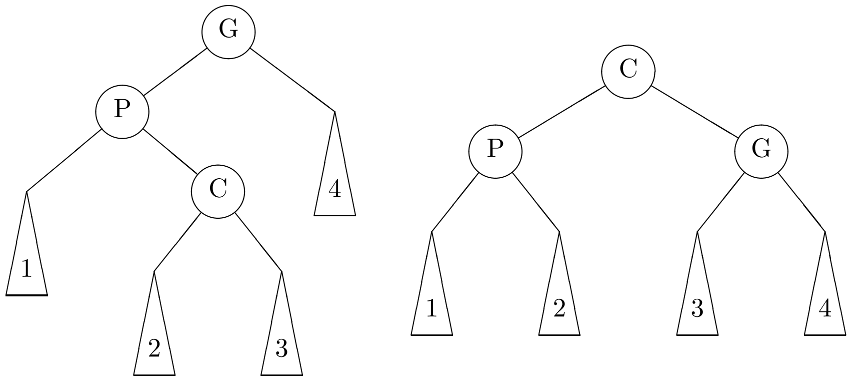

Figure 1: Zig-zag rotation.

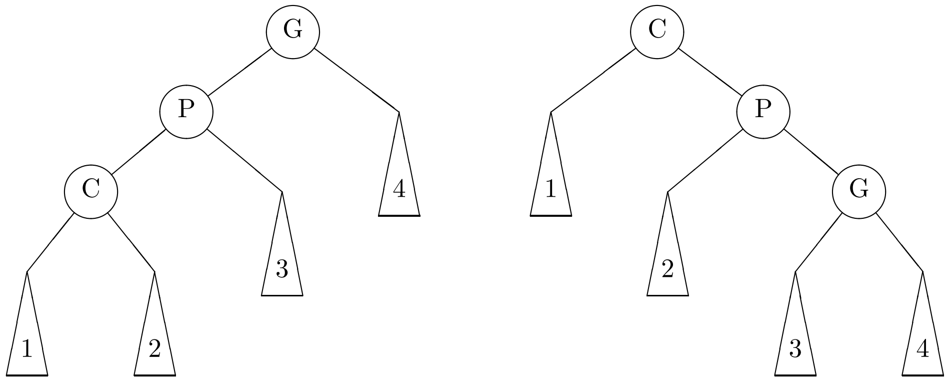

Figure 2: Zig-zig rotation.

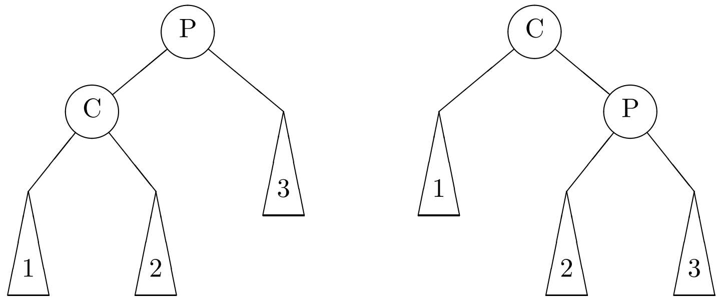

Figure 3: Single rotation.

and where a node is right child of the parent, and the parent is the right child of the grandparent. An example of zig-zig rotation is shown in Figure 2, where a node C with a parent P and a grandparent G are zig-zigged.

The final class of rotation is where there is no grandparent, that is, where the node to be rotated is the child of the root. The process of *single rotation* is shown in Figure 3.

Bottom-up splaying is an intuitive method of applying rotations. In bottom-up splaying, the tree is first searched. After finding a matching node—or the point of insertion for a new value—rotations are repeatedly applied to the node to bring it to the root of the tree. In the case of a search where a value is not found, the last non-null node in the tree is splayed. In fast implementations, an additional pointer from a node to its parent is stored to permit reorganisation without storing the search path. This pointer allows significant savings, since splay trees require significant local adjustments both during construction and—in contrast to balanced structures—with each access. We report experiments with bottom-up splaying in Section 6; heuristic implementations of bottom-up splaying are described in Section 4.

Sleator and Tarjan also describe a *top-down* splaying scheme that requires only one tree traversal to both search and reorganise. Top-down splaying works as follows. First, two empty trees, a *left* and *right* tree, are initialised. Second, the tree is traversed two nodes at a time, breaking the links between the nodes. Third, with each two-node traversal, any subtree known to be less than the current node is added to the bottom-right of the left tree, and any subtree known to be greater than the current node is added to the bottom-left of the right tree. Last, if the two nodes are traversed left-left or right-right, then a rotation is performed before adding the subtree to the left or right trees. Details are provided by Sleator and Tarjan [20]; we report experiments with top-down splaying in Section 6.

Splaying has found application in areas where amortised performance is important, including in data compression [7, 10], sorting [14], and index construction [22, 24]. In addition, extended n-ary splay trees and variants have been proposed and compared to static n-ary trees, with similar amortised performance [12, 19]. We explore in Section 4 heuristics for fast splaying for applications that manage large vocabularies.

3 RANDOMISED SEARCH TREES

Randomised search trees [18] are another variation of binary search trees that employ rotation heuristics to achieve expected bounds comparable to the amortised bounds of the splay tree. (We refer to randomised search trees as RTs in the rest of this paper.) RTs store a weight or *priority* in each node that—while having additional storage cost—reduces significantly the number of required rotations to reorganise the tree compared to a splay tree. Much as in a splay tree, rotations are used to bring more frequently accessed nodes nearer the root of the tree and, therefore, RTs are a candidate structure for accumulating large vocabularies. We explain how priorities are used to control rotations in this section.

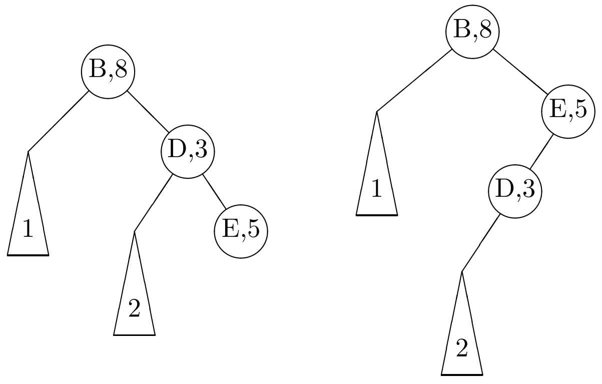

Figure 4: Insertion of node E with a priority 5 into a treap. The treap is shown on the left immediately after the insertion of E, where the keys are in-order correctly but the heap property is violated. The treap shown on the right is after reorganisation so that both the heap property and the key ordering property are correct.

RTs are based on so-called *treaps*, a term coined to convey both the tree and heap nature of the structure. The underlying structure of the treap is a BST, with the usual property that the nodes are in-order with respect to the keys; therefore, when used for vocabulary management, an in-order traversal produces a sorted list of words in the same way as a BST or splay tree. However, the treap also maintains *heap order* so that the priority value stored in a node is always less than or equal to its parent. If the heap priority is violated by, for example, inserting a new node with a high priority into the tree, then single rotations are applied to the new node until a parent is assigned that has a higher priority (or the new node becomes the root of the tree).

An example treap is shown in Figure 4. The treap is shown in the left of the figure immediately after the insertion of a new value "E" with a priority of 5. As in a BST, the node is first inserted as a leaf node so that the ordering of keys is maintained. However, as the priority of the new node is 5 and this is greater than the parent node "D" priority of 3, the tree must be reorganised to maintain the heap ordering using a single rotation. The treap after application of the rotation is shown in the right of the figure, where the heap property is now regained without violating the tree property.

An RT is a treap where the priorities are independently assigned, identically distributed random values[1]. Random priorities are assigned when a node is inserted, and a new priority is also generated each time a node is accessed as the result of a successful search. When searching, if the new priority generated is greater than the current priority of the node, the new priority is assigned to the node and the tree rotated so that the treap property is maintained.

If a node is accessed frequently, its priority will be the maximum of the values generated at each access. Therefore, the expected depth of a node in the RT will be of $O(\log(1/f))$ where f is the access frequency of the node. In other words, the depth of a node in the RT is inversely proportional to the number of accesses of that node, ascribing to the structure the self-adjusting property needed for accumulation of skewed data sets.

Several heuristics have been proposed [13, 18] to reduce time or space bounds of RTs. For example, parent pointers are proposed to reduce reorganisation costs and permit implementing an efficient non-recursive bottom-up algorithm. For space reduction, three heuristics to avoid explicitly storing priorities are proposed: first, it is suggested a hash function be used to generate a pseudo-random number using the node key as an input for generating a priority as required; second, nodes can be stored in an additional pointer array in a random order and the array position used as the priority; and, last, the node priority can be assigned as a function of the size of the

[1]Ideally, the random values should also be continuous, that is, there should be no nodes with identical values. However, in practice, the expected performance is not affected if only a few nodes contain identical values.

node's subtree. We do not discuss space heuristics in detail here and have not tested space-efficient RTs; however, we would expect that the space saving would be offset by the additional time cost of generating priorities.

We report experiments with RTs in Section 6.

4 HEURISTICS FOR FAST SPLAYING

Sleator and Tarjan [20] propose heuristics for splaying that are intended to reduce the restructuring required, while maintaining the amortised cost of accesses. We explore one class of these heuristics, which Sleator and Tarjan refer to as *semi-adjusting*, and we explore a novel class of heuristics which we call *limit-splaying*. These heuristics involve splaying only sometimes when a search or insertion occurs. Our heuristics can be applied easily to bottom-up splaying; it is unclear how they might be applied to top-down schemes.

The first limit-splaying technique is to splay after each n insertions or accesses, for a fixed value n. We call this heuristic PERIODIC-ROTATION. The rationale of this approach is to reduce the total rotation cost, and in particular unnecessary churning of nodes near the root of the tree, while approximately maintaining the clustering of frequently accessed nodes near the root. Without this heuristic, the tree is splayed on each access; with the heuristic each node in a large collection will be splayed on average n times less. For small values of n, PERIODIC-ROTATION approximates splaying, while for large values of n, PERIODIC-ROTATION approximates a BST. A similar variant, where rotations are performed when a probability is exceeded was recently proposed by Furer [5]; this method has not been tested in practice.

The second heuristic, a semi-adjusting scheme proposed by Sleator and Tarjan, is to splay until a threshold m is reached, and then to stop splaying. We call this STOPSPLAY. Possible choices of a threshold m include until a fixed number of words are processed, a percentage of the collection has been inserted, or a threshold is reached and the tree is approximately balanced; fast measurement of balance in incomplete trees is a difficult problem that we do not discuss here. The rationale is again to reduce the overall cost of rotation through ceasing rotation when the tree has frequently accessed nodes close to the root. For small values of m, STOPSPLAY should approximate a BST. For large values of m, where splaying is maintained for a significant fraction of the insertions, STOPSPLAY should closely approximate splaying. The drawbacks of this approach—as noted by Sleator and Tarjan [20]—are that splaying may be stopped when the tree is inefficiently organised or, after stopping splaying, the access frequencies to words may change rendering the tree an inefficient structure.

The third heuristic is a limit-splaying heuristic to maintain counters of node accesses for each node and to splay a node when a threshold l is exceeded; this approach is related to RTs, but counters are used instead of priorities, and nodes are splayed to the root of the tree on each access. There are many possible choices for threshold. For example, the threshold may be a fixed value l, or relative to the total frequency of words processed in the collection. Additionally, after splaying, the node may be marked for splaying on each access, or the counter l reset. We call this scheme COUNT.

As in the previous schemes, the aim of COUNT is to reduce the overall cost of rotation while maintaining clustering of frequently accessed nodes near the root. The difference is that only frequently-accessed nodes are brought to the root. For low values of l, the scheme approximates splaying; for large values of l the scheme approximates a BST. A disadvantage of this scheme is the necessity of storing a counter for each node, a cost that depends on the maximum value to be stored; a simple space-efficient variant is to store a single-byte counter and splay when this value reaches 256 and reset to 0.

A second, distinct class of bottom-up splaying heuristic is discussed in detail by Sleator and Tarjan [20] and more recently by Furer [5]. In this variant—which is referred to as *semi-splaying*—an accessed node is not restructured so that it becomes the root but, rather, is moved only part of the way towards the root. We have experimented with variants of semi-splaying, such as splaying half-way to the root, semi-splaying one level, and with metrics based on frequency. In all cases, we have found that semi-splaying performs worse than all other variants described in this paper, including RTs. For this reason, we do not report detailed results here.

The great strength of splay trees, in contrast to BSTs, is their worst case. For this application, in the worst case the average access cost is logarithmic in the number of distinct words, regardless of the characteristics of the text. In contrast, in BSTs the average cost of individual accesses can be linear in the number of distinct words, a problem that can easily arise in practice, if a text collection commences with a sorted list of words for example.

None of the heuristics described above preserve the good worst-case characteristics of standard splay trees. With the PERIODIC-ROTATION scheme, for example, if every nth word occurrence is a new word that is greater lexicographically than all previous words, and if all other occurrences are words that are less lexicographically than all previous words, search costs will be close to the worst case of a BST. However, we would not expect such patterns to arise in practice. Even in cases where such patterns were observed over a region of the text, the impact on total costs would only be serious if the region were very extensive.

5 OTHER TECHNIQUES

In our experiments, we compare RTs, splay trees, and heuristic splaying approaches to other candidate data structures. Specifically, we have implemented a red-black tree, an efficient bit-wise hash table, and a BST. We discuss these approaches briefly here. There are, of course, many other structures that could be used for this task, such as other balanced tree structures, or skiplists.

Skiplists would be unlikely to be efficient in this application, because as noted by Pugh [15] they require more key comparisons than do splay trees. Additionally, balanced tree structures are not the focus of this work and, as such, we have implemented only red-black trees as a comparison point. As we discuss later in Section 6, we would not expect balanced trees to be well suited to accumulating skewed data sets; balanced trees are as likely to have rare words stored in the leaves as common words, and no adaptation to the locality or probability of occurrence of words occurs.

Red-black trees [2, 8] are a variation on BSTs, with an additional bit stored in each node to "colour" the node either red or black. The significant difference is that the tree is balanced so that the depth of a node at the leaf is at most $2 \log n$ for a tree of n nodes. Individual access time to nodes is guaranteed to be $O(\log n)$ by rotating nodes on insertion so that three properties are maintained:

1. Every path from the root to a leaf has the same number of black nodes
2. No red parent node has a red child
3. The root node is black

All tree maintenance is a constant overhead: there are at most two rotations per insertion, and m key insertions require at most m changes of node colour. Moreover, no rotations occur as a result of searches.

The hash table we have implemented is efficient for in-memory management of vocabularies. The hash table is an array of a fixed number of slots, with chained pointers from each array slot to a linked-list of nodes containing the keys stored in the slot (and a pointer to the next node). Words are allocated to slots (and then stored in the linked lists chained from the array slots) by applying a hash function to the word.

The choice of hash function is crucial to the performance of the hashing scheme. We use an efficient "bit-wise" hashing function that behaves much like a traditional "radix" hash function [17] but avoids the use of computationally-expensive modulo and multiplication operations. The bit-wise scheme uses only shift and exclusive-or operations [16].

Notionally hashing—like BSTs—has a worst case of linear access cost on each search, should each word hash to the same slot. However, this worst case is absurdly improbable [6]; for chained tables, the longest chains are at worst a few times greater than the average. For example, the likely worst case in practice of one million strings hashed into a chained hash table of one million slots with a uniform hash function is a chain of around 10 nodes [16], with an average search cost of around 1.3 comparisons.

We would not expect unbalanced BSTs to be generally suited to accumulating vocabularies. The vocabulary of a text collection can vary, for example if the language of the documents changes. In such cases, in some documents none of the words searched for would be close to the root. In contrast, self-adjusting trees should rapidly adjust to the new vocabulary; indeed, most individual documents to some extent have an idiosyncratic vocabulary, such as the places or names referred to in a particular news item, and a self-adjusting tree should adjust to such local properties. As we discuss in the next section, our test data includes documents in several languages, predominantly English but with many documents in German, and thus this problem is a factor in the performance of the BST.

However, in the case of text collections all in one language, the BST can be expected to perform reasonably well. Statistically, common words such as "the" or "of" are likely to occur early in the collection, and thus will be high in the tree. Indeed, for this reason techniques such as red-black rebalancing are likely to be ineffective, since their average effect is to move common words down the tree.

The choice of string comparison function was another important factor in the efficiency of all the schemes tested in this paper. The standard `strcmp()` string-comparison function provided with both Solaris and Linux proved to be highly inefficient; replacing it with a simple iterative function with the same specification as standard `strcmp()` yielded, for example, a reduction in total time of around 20% for PERIODIC-ROTATION-based splaying[2]

6 RESULTS

We tested the data structures described by applying them to a collection of around 1 Gb of world-wide web text data derived from the TREC [9] Very Large Collection. TREC is an ongoing international collaborative experiment in information retrieval sponsored by NIST and ARPA. In all, the collection contains 171,397,973 word occurrences[3] of 1,973,187 distinct words in 616,944 documents, with an average of 3.2 new words per document. This collection is typical of a collection managed by a text database system: it shows the well-known Zipfian distribution of word frequencies [23]; it has occurrence locality [24], where many words occur frequently in small regions of the data; and new words appear with an average frequency of around three per document.

All experiments are carried out on an Intel Pentium II 266 MHz dual processor machine with 320 Mb of main-memory running the Linux operating system under light load, where the machine is otherwise largely idle. Reported times are for the complete process of reading and parsing text and other housework, as well as processing of the data structures themselves. The time required for the housework alone is approximately 264 seconds.

6.1 SPLAY TREE HEURISTICS

Figure 5 shows the results of applying our PERIODIC-ROTATION scheme to the TREC data. The effect of using PERIODIC-ROTATION is twofold; the number of rotations performed drops dramatically as rotation frequency n is increased, while the number of string comparisons slowly rises. For small values of n the combined effect is to greatly reduce costs. For example, splaying after each 11th tree traversal can reduce the costs to 80% of efficiently-implemented bottom-up splaying. This improvement in speed can be attributed to the around 90% reduction in total rotations

[2]Standard `strcmp()` has one assignment, one pointer subtraction, one inequality test, and a pointer increment per character comparison. Our implementation has one equality test, one inequality test, and two pointer increments per character comparison, and one pointer subtraction per string. Our scheme reduces pointer subtractions significantly and saves an assignment per character (this saving is significant compared to the per-character cost of an extra pointer increment and equality test).

[3]Non-words, including punctuation, special characters, SGML and HTML markup, sequences of ASCII characters containing more than two digits or beginning with a digit, and sequences of ASCII characters containing special characters were removed.

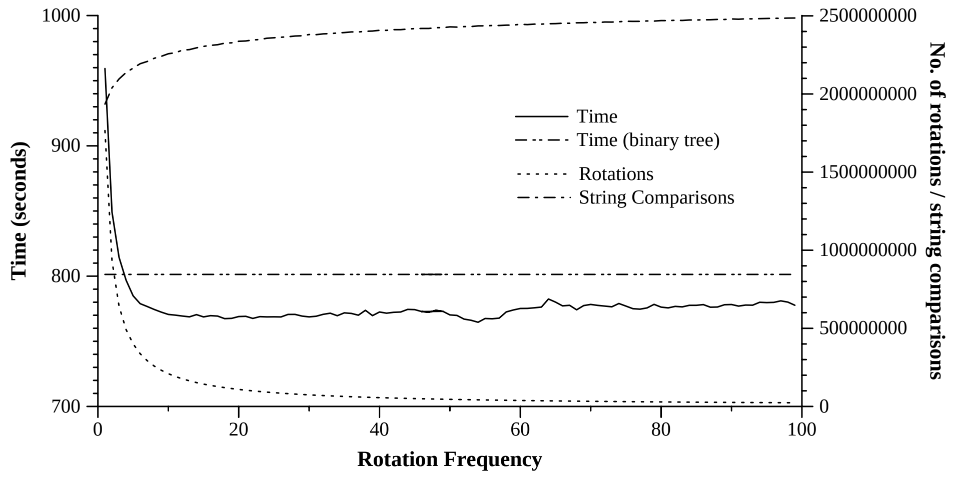

Figure 5: Rotation heuristic for splaying. In this approach, a node found or inserted is splayed after each n tree accesses for a fixed value of n. Rotation each $n = 11$ times or more reduces the time cost to around 80% of unmodified bottom-up splaying.

with only a moderate 17% increase in string comparisons. This improved speed with PERIODIC-ROTATION remains roughly consistent for all values greater than $n = 11$; for high values of n total rotations are reduced further, a benefit balanced by the cost of more string comparisons as the tree becomes less balanced and has poorer adaptation to local changes in the frequency of words.

Figure 6 shows the performance of the STOPSPLAY scheme, which is less effective than PERIODIC-ROTATION. In this approach, we splay on each access as normal until m documents have been processed, and then cease splaying. In this figure, the horizontal axis is the value of m, the vertical is the total time to process the full collection for that value of m. The rotation cost is dependent on the threshold m and, for low values of m, the rotation cost is insignificant. However, the time cost of processing the collection closely follows the total string comparison cost, which varies significantly depending on when splaying is stopped. By using a fixed value of m splaying may or may not be stopped when common words are clustered near the root; as can be seen, similar values of m lead to highly different total processing times. For some small values of m, STOPSPLAY works slightly better than a BST but for all values of m greater than around 12% of the collection size STOPSPLAY is worse. The STOPSPLAY scheme might be improved by stopping and starting splaying depending on two thresholds, or measuring completeness and balance of the tree. However, even with modifications, we believe that it is unlikely that STOPSPLAY will be better than the PERIODIC-ROTATION scheme.

As shown in Figure 7, the COUNT heuristic is unsuccessful in reducing the costs below that of a BST. To implement COUNT, we have added a counter to each node in the tree, where a node is splayed when the count exceeds a threshold l. As in other schemes, a reduction in rotation costs and an increase in string matching costs are the result of less splaying. The COUNT scheme does not work well because new words are inserted as new leaf nodes and are not splayed until the threshold l is exceeded; this prevents the splay tree from fast adaptation to changes in word occurrence frequencies where, for example, a new word will appear frequently only for a few documents.

However, we have found in preliminary experiments that variations of COUNT that compare counts as a precondition to rotation—much like the preconditions using priorities in RTs—yield some improvements in performance. We have observed similar improvements through combinations with the PERIODIC-ROTATION approach. We plan to experiment further with these composite approaches. A drawback of this class of approaches are the small costs of maintaining and testing counters.

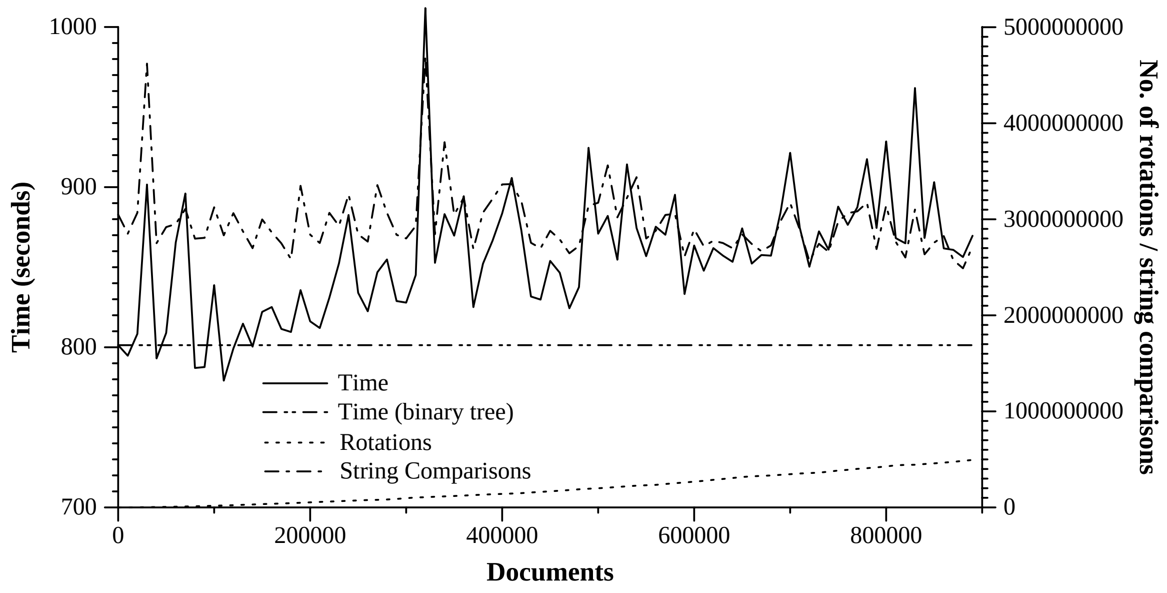

Figure 6: Stopsplay heuristic for splaying. In this approach, the tree is splayed normally until m documents have been processed, after which splaying is stopped. In general, the stopsplay heuristic is inconsistent, depending on tree completeness and balance, and does not offer any significant performance improvements.

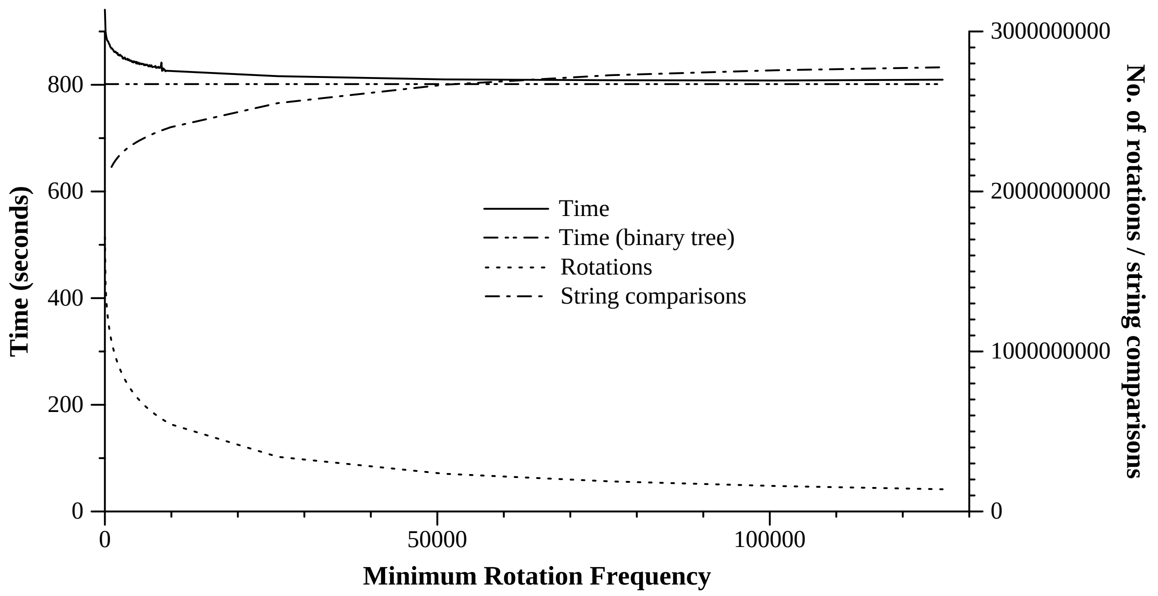

Figure 7: Count heuristic for splaying. In this approach, a node is splayed only when its frequency exceeds a threshold l. This heuristic does not work well, as it removes a significant benefit of splaying being able to adapt to the locality word frequencies.

Table 1: Results with top-down and bottom-up splaying, heuristic bottom-up splaying, a red-black tree, an RT, a BST, and a hash table on five 1 Gb web document collections. The heuristic splaying scheme used is the best available scheme, that is, periodic-rotation after every 11th insertion. A paired t-test at the 5% significance level confirms the relative ordering of the schemes from slowest (top-down splaying) to fastest (hashing).

Heuristic	Mean Time Seconds (S.D.)	Mean Rotations (S.D.)
Top-down splaying	1118.8 (52.6)	625×10^6 (29×10^6)
Bottom-up splaying	1003.3 (45.1)	$1,841 \times 10^6$ (84×10^6)
Randomised Search Tree (RT)	976.4 (33.9)	59×10^6 (3×10^6)
Red-black tree	843.0 (39.6)	1×10^6 (0×10^6)
Binary Search Tree (BST)	808.2 (35.8)	—
Periodic-rotation	790.0 (35.8)	200×10^6 (8×10^6)
Hashing	669.4 (42.2)	—

6.2 COMPARISON OF STRUCTURES

11cm

Overall results comparing self-adjusting structures to a BST, red-black tree, and hashing are shown in Table 1. In these experiments, we present average results with five different text collections of around 1 Gb in size derived from the TREC [9] Very Large Collection web data. The hash table is moderate in size, containing 220,000 slots. All self-adjusting trees tested use parent pointers for fast tree reorganisation, and all structures use our implementation of `strcmp()`. We use the C library function `rand()` to generate priorities in the RT[4]. For our results, a paired t-test supports at the 5% level the hypothesis that the differences in speed shown in the table for the different schemes are statistically significant.

The results support the conclusion that BSTs are faster than standard splaying for storing large lexicons. The BST has performed surprisingly well, despite the presence of a significant volume of non-English text in the test data. It appears that common words are indeed stored near the root, by virtue of their occurrence early in the test data, and the cost of searching deeper in the tree, on average, than the splay tree is more than offset by the lack of rotations. Each rotation (zig-zig or zig-zag) requires about six comparisons and ten assignments, a significant cost.

The results also show that RTs are 3% faster than bottom-up splaying and 21% slower than a BST. Relative to bottom-up splaying, the RT rotates 30 times less often but uses computationally expensive random-number generation. Additionally, because splaying reorganises a tree significantly—as opposed to no reorganisation or some local reorganisation—splaying is likely to make less effective use of the CPU cache for reading and requires more writes to main-memory.

The results also illustrate why balanced-tree schemes such as red-black trees are less efficient for this application. For PERIODIC-ROTATION with $n = 11$, the average number of string comparisons per access is 14.4. However, there are almost two million distinct strings, and thus in a balanced red-black tree (where common strings have the same likelihood as other strings of being held at the leaves) the average number of string comparisons is over 20.

A surprising result is that in practice—and in contrast to the analysis of Sleator and Tarjan [20] and popularly held wisdom [21]—we have found top-down splaying is slower than bottom-up despite the smaller number of rotations in the former. While top-down splaying has at most one rotation per accessed node and bottom-up splaying has exactly two, the slowness of top-down splaying is due to the reorganisation costs associated with pointer reassignment. In addition, despite the costs in bottom-up splaying of the additional tree traversal required per reorganisation, parent-pointers reduce pointer assignment costs significantly. Interestingly, possibly because of

[4]While other random-number generators may be more efficient and generate values that are closer to random [11], we have found empirically that other random-number generators do not markedly affect the overall speed of vocabulary accumulation.

our use of parent pointers, our results are in contrast to a previously-reported empirical study by Pugh [15] that found that bottom-up splaying was less efficient than top-down splaying. Another contrast was that Pugh's experiments used integer keys. If strings are used as keys, the cost of key comparison is a much greater component of total costs, outweighing savings that might be available.

Our results are also in contrast to those of Gopal and Bell [4], whose comparative study of splay trees and other tree structures found that that bottom-up splay trees are many times slower than BSTs and that top-down splaying is faster than bottom-up splaying. These results, however, were determined in a very different test environment. The data set itself was small enough that the whole 48 Kb tree could fit into the cache of a typical CPU, and was the case in Pugh's work based on integer keys. Another crucial difference was that real data does not behave like the randomly-generated data used by Gopal and Bell; for example, in real data, the occurrences of search terms, even real search terms, cluster. Their results are not applicable to large data sets or to sets of string data.

Of the heuristics we have explored for improving bottom-up splaying, the only one that is consistently successful is the PERIODIC-ROTATION scheme: it is more than 27% faster than efficiently-implemented bottom-up splaying but just 3% faster than a BST. However, the worst-case of an unbalanced BST is a practical possibility, and thus for some applications splaying is clearly preferable.

Hashing is the fastest of the schemes we tested for index construction, and is more than 15% faster than periodic rotation; the improvement in speed with hashing is due to the reduction in the number of string comparisons, as well as due to use of the efficient shift-based hash function. We have recently investigated variations of hashing for accumulating vocabularies and reported these results elsewhere [26]. With other string hashing functions, all versions of trees were faster. However, a disadvantage of hashing is that sorting of the vocabulary is required to support, for example, range or prefix queries; in applications where sorted access is required, our PERIODIC-ROTATION scheme is preferable.

7 Conclusions

We have investigated self-adjusting trees for storing large lexicons in text database index construction. In experiments on large text collections we have shown that a heuristic splay tree—where the splay tree is splayed periodically rather than on each access—is on average around 27% faster than efficient bottom-up splaying. Over five separate text collections we have found several surprising results: first, top-down splaying is slower than bottom-up splaying in practice; second, bottom-up splaying is around as fast as a self-adjusting randomised tree (RT), but in general around 25% slower than a BST; and, last, the most efficient heuristic splaying scheme is only 3% faster than a BST.

Despite the skew distribution of terms in our data sets, in contrast to the conjecture of Sleator and Tarjan [20] we have not found that "[standard] splay trees are as efficient on any sufficiently long sequence of accesses as any form of dynamically updated search tree", but that they are rather less efficient than an unbalanced BST. In addition, we have shown that hashing is faster than efficient trees for index construction, but only with an appropriate choice of hash function.

We conclude that for index construction either our periodic rotation splay tree or hashing should be used. While a BST is nearly as efficient as heuristic splaying, the possibility of worst-case performance means that it should not be used; such worst-case performance is unlikely with heuristic splaying.

Acknowledgments

We thank Neil Sharman for valuable discussions on the use of hashing for index construction in the SIM database system. We also thank the reviewers for their valuable comments. This work was supported by the Multimedia Database Systems group at RMIT University.

References

[1] S. Albers and J. Westbrook. Self-organizing data structures. In Amos Fiat and Gerhard Woeginger, editors, *Online Algorithms: The State of the Art*, pages 31–51. Springer-Verlag, 1998.

[2] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, November 1972.

[3] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[4] J. Bell and G.K. Gupta. An evaluation of self-adjusting binary search tree techniques. *Software—Practice and Experience*, 23(4):369–382, 1993.

[5] M. Furer. Randomized splay trees. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 903–904, Baltimore, MD, 1999.

[6] G. Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM*, 28(2):289–304, 1981.

[7] D. Grinberg, S. Rajagopalan, R. Venkatesan, and V.K. Wei. Splay trees for data compression. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 522–530, San Francisco, California, 22–24 January 1995.

[8] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Long Beach, Ca., USA, October 1978. IEEE Computer Society Press.

[9] D. Harman. Overview of the second text retrieval conference (TREC-2). *Information Processing & Management*, 31(3):271–289, 1995.

[10] D.W. Jones. Application of splay trees to data compression. *Communications of the ACM*, 31(8):996–1007, August 1988.

[11] G. Marsaglia. Remarks on choosing and implementing random number generators. *Communications of the ACM*, 36(7):105–110, July 1993.

[12] C. Martel. Self-adjusting multi-way search trees. *Information Processing Letters*, 38(3):135–141, May 1991.

[13] C. Martínez and S. Roura. Randomized binary search trees. *Journal of the Association for Computing Machinery*, 45(2):288–323, March 1998.

[14] A. Moffat, G. Eddy, and O. Petersson. Splaysort: fast, versatile, practical. *Software Practice and Experience*, 26(7):781–798, July 1996.

[15] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[16] M.V. Ramakrishna and J. Zobel. Performance in practice of string hashing functions. In *Proc. International Conf. on Database Systems for Advanced Applications*, pages 215–223, Melbourne, Australia, April 1997.

[17] R. Sedgewick. *Algorithms in C: Parts 1–4: Fundamentals, data structures, sorting, searching.* Addison-Wesley, Reading, MA, USA, 1998.

[18] R. Seidel and C.R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[19] M. Sherk. Self-adjusting k-ary search trees. *Journal of Algorithms*, 19(1):25–44, July 1995.

[20] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[21] M.A. Weiss. *Data Structures and Algorithm Analysis in C.* Addison-Wesley, 2nd edition, 1997.

[22] H.E. Williams and J. Zobel. Indexing and retrieval for genomic databases. *IEEE Transactions on Knowledge and Data Engineering.* (to appear).

[23] I.H. Witten and T.C. Bell. Source models for natural language text. *International Journal on Man Machine Studies*, 32:545–579, 1990.

[24] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

[25] J. Zobel. How reliable are the results of large-scale information retrieval experiments? In R. Wilkinson, B. Croft, K. van Rijsbergen, A. Moffat, and J. Zobel, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 307–314, Melbourne, Australia, July 1998.

[26] J. Zobel, S. Heinz, and H.E. Williams. In-memory hash tables for accumulating text vocabularies. Manuscript in submission, 2000.