SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(4), 369–382 (APRIL 1993)

An Evaluation of Self-adjusting Binary Search Tree Techniques

JIM BELL AND GOPAL GUPTA
Department of Computer Science, James Cook University, Townsville, Queensland 4811, Australia

SUMMARY

Much has been said in praise of self-adjusting data structures, particularly self-adjusting binary search trees. Self-adjusting trees are most suited to skewed key-access distributions as the techniques attempt to place the most commonly accessed keys near the root of the tree. Theoretical bounds on worst-case and amortized performance (i.e. performance over a sequence of operations) have been derived which compare well with those for optimal binary search trees. In this paper, we compare the performance of three different techniques for self-adjusting trees with that of AVL and random binary search trees. Comparisons are made for various tree sizes, levels of key-access-frequency skewness and ratios of insertions and deletions to searches. The results show that, because of the high cost of maintaining self-adjusting trees, in almost all cases the AVL tree outperforms all the self-adjusting trees and in many cases even a random binary search tree has better performance, in terms of CPU time, than any of the self-adjusting trees. Self-adjusting trees seem to perform best in a highly dynamic environment, contrary to intuition.

KEY WORDS: Binary trees Splay trees AVL trees Self-adjusting trees

INTRODUCTION

The binary search tree (BST) is a commonly-used data structure for storing and retrieving records in main memory because it guarantees logarithmic cost for various operations as long as the tree is balanced. It is therefore not surprising that techniques that maintain balance in BSTs have received considerable attention over the years. The most popular balancing technique is the AVL or height-balancing technique[1] which performs local balancing whenever the height-balance is violated. Other local balancing techniques such as bounded balance[2] and weight balance[3] have also been suggested. It is of course possible to balance the entire tree at one time and obtain a complete tree or a tree that is close to being complete.[1,4,5]

The above balancing techniques are designed to be efficient when all keys in the tree are expected to be searched with equal probability. For skewed key-access distributions, one may build an optimal binary search tree if the access frequencies are fixed and known in advance. Building an optimal tree requires $O(n^2)$ time and space, and, although a near-optimal tree can be constructed in $O(n)$ time,[6] the technique becomes quite inefficient if used every time an insertion or deletion is made to the tree. Furthermore, access frequencies are usually not known in advance,

0038–0644/93/040369–14$12.00

Received 3 August 1991
Revised 14 October 1992

and are sometimes not fixed. In such situations, it has been suggested that a self-adjusting technique for binary search trees is likely to be most efficient. Several such techniques have been suggested in the literature. In this paper we consider three techniques that are called splaying, exchange and ‘move to root’, which all promote frequently-accessed keys either directly or indirectly to the root. The methods are described in the next section.

The primary aim of this paper is to compare these three self-adjusting binary search tree structures with the classical AVL tree and the random BST.

The rest of the paper is organized as follows. The three self-adjusting binary search trees to be compared are introduced. This is followed by a description of the methodology used in the comparison of the techniques. Finally, we present our results, and the conclusions drawn from these.

SELF-ADJUSTING BINARY SEARCH TREES

The self-adjusting BST structures promote frequently-accessed keys toward (or to) the root by modifying the tree at every access, and usually at every insertion and deletion.

Some of the early work on self-adjusting binary trees was reported by Bitner[8] and Allen and Munro.[7] These works examined a technique that exchanged an accessed key with its parent by performing a single rotation. Two symmetric rotations are possible, as shown in Figure 1(a). We refer to this method as *exchange*. Figure 2(a) shows a binary search tree before accessing 25. Figure 2(b) shows the tree after accessing 25 using the exchange method. Note that the number of comparisons required to find the key 25 after accessing that key has been reduced by one.

Early research also examined a strategy that moved the key being accessed to the root by a series of rotations as above. We shall refer to this technique as *move to root* (MTR). Figure 2(c) shows the resulting tree when key 25 is accessed in the tree of Figure 2(a) using MTR. Allen and Munro[7] have analysed the performance

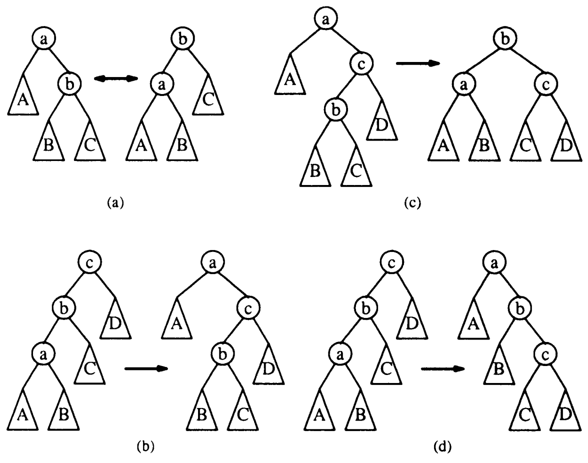

Figure 1. Rotation operations: (a) single rotations; (b) and (c) rotation sequences used for MTR; (c) and (d) rotations pairs used in splaying

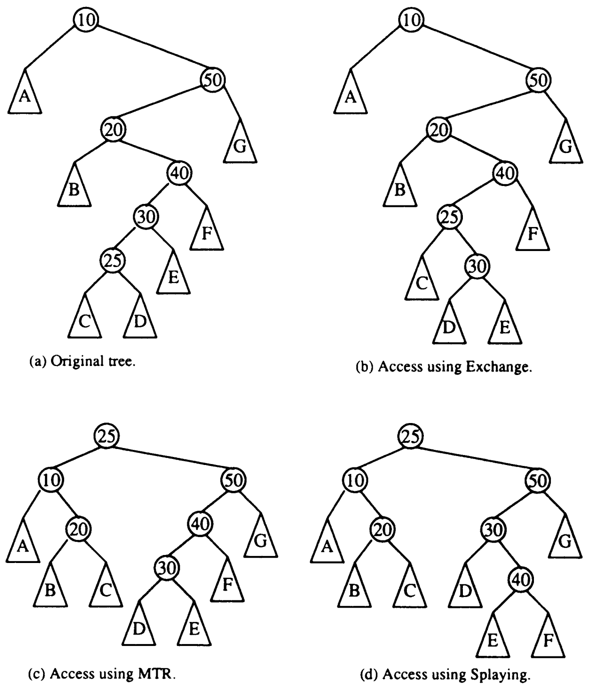

Figure 2. Accessing key 25 using self-adjusting tree methods

of simple exchange and the MTR technique, and have shown that if there are n keys with search probabilities that are uniformly distributed, the average search cost for the exchange algorithm approaches $\sqrt(\pi n)$ and for MTR technique the cost is approximately $1{\cdot}3863\lg(n)$.* Allen and Munro suggest that the exchange method should never be used because of its poor performance in the above case.

Sleator and Tarjan[9] introduced the splay tree. We shall discuss two different methods of splaying, *top-down splaying* (TDS) and *bottom-up splaying* (BUS) both of which move an accessed key to the root as with MTR, but using different techniques.

Sleator and Tarjan presented an amortized analysis which showed that insertions, deletions and searches have a cost bound of $O(\lg(n))$ over a sequence of worst-case operations. It is shown that over a sufficiently long sequence of accesses, the performance of splay trees is within a constant factor of the performance of an optimal tree.

Splay trees perform rotations during all operations. Each accessed or inserted key is moved to the root, and the predecessor/successor of a deleted or unsuccessfully-searched key is promoted to the root. In the case of the exchange method and the MTR, References 7 and 8 do not specify what should be done at insertion or deletion. To be consistent, we have assumed that rotations are carried out in a similar way to that for the splay tree at each insertion and deletion.

Bottom-up splaying promotes a key to the root by pairs of rotations that consider the position of the accessed node relative to its parent and grandparent. Two pairs

* Throughout this paper, lg stands for the logarithm to base 2.

of symmetric rotations are used in BUS. The rotation in Figure 1(c) is performed if the accessed node, *b*, is the left (right) child of the right (left) child of its grandparent. This is the same sequence of rotations as would be performed if MTR were used. The difference between MTR and BUS is seen by comparing Figures 1(b) and 1(d). Figure 1(d) shows the rotations performed if the accessed node, *a*, is the left (right) child of the left (right) child of its grandparent.

Figure 1(b) shows the corresponding sequence of rotations which would have been performed using MTR. If the node is a child of the root node, the appropriate single rotation (Figure 1(a)) is performed to make *x* the root. Figure 2(d) shows the effect of accessing 25 in the tree of Figure 2(a) using splaying. The first splay operation causes the splayed tree of Figure 2(d) to differ from the MTR tree in Figure 2(c).

Top-down splaying effectively splits the tree along the search path from the root to the accessed key. During the splitting process two temporary trees are formed. *L* contains all keys less than the search key and *R* contains all keys greater than the search key. As we proceed along the search path to the required key, the following steps are performed repeatedly:

1. If the accessed key is the left (right) child of the current node then the current node and its right (left) subtree is appended to *R* (*L*) as the leftmost (rightmost) subtree.
2. If the accessed key is in the left-left (right-right) subtree of the current node then a right (left) single rotation is performed at the current node. Then the new current node and its right (left) subtree is appended to *R* (*L*) as the leftmost (rightmost) subtree.
3. If the accessed key is in the left-right (right-left) subtree of the current node we proceed in two steps. First, remove the current node and its right (left) subtree, appending these to the far left (right) of the *R* (*L*). This leaves the left subtree of the current node remaining. Secondly, remove the root and left (right) subtree of that tree and append them to the far right (left) of *L* (*R*). What remains is the subtree containing the accessed key.
4. If the current node contains the accessed key, then append its left and right subtrees as the rightmost and leftmost subtrees of *L* and *R* respectively. Make *L* and *R* the left and right subtrees of the current node and stop.

A full description of top-down splaying is provided by Sleator and Tarjan.[9] TDS has the same amortized performance bounds as BUS; however, no link to parent nodes (either explicit or implicit) is required and fewer rotations are performed (as shown later). Sleator and Tarjan comment that TDS performs better than BUS for basic tree update and search operations, but provide no evidence.

To evaluate the effectiveness of the three self-adjusting BST structures, we compare them with the random BST and the AVL tree. Insertions, deletions and retrievals on a random BST do not require any rebalancing transformations, as no attempt is made to keep the tree balanced. The average cost of searching a randomly built tree of size n is approximately $1{\cdot}38\lg(n)$, although such trees can have a worst-case height of n.[1] This tree structure is seldom used in practice, due to its poor worst-case performance.

The AVL or height-balanced BST was introduced by Adel'son-Vel'skii and Landis in 1962.[1] AVL trees perform rotations when necessary during insertions and deletions to maintain the constraint that the heights of the left and right sub-trees of any node differ by at most 1. This constraint ensures that the height of the tree never exceeds $1{\cdot}44\lg(n)$, and hence limits the worst-case search cost. Empirical evidence indicates that the AVL tree has a mean search cost of $\lg(n) + O(1)$ under a uniform access distribution.[1] Two different rotation operations are used to maintain the height-balance in an AVL tree (as shown in Figures 1(a) and 1(c)).

We now summarize the characteristics of the five methods in Table I. An entry in the table indicating that rotations always occur excludes the case in which the key is located at the root.

COMPARISON METHODOLOGY

As far as possible, the techniques were coded in an unbiased fashion using pointer reversal when needed instead of recursion during tree operations. Pointer reversal is often used for two reasons. First, pointer reversal requires only a constant memory overhead, whereas recursion requires memory proportional to the length of the access path to the key. Secondly, as some of the techniques require a node to access its parent, pointer reversal allows this to happen without having an explicit parent pointer for each node. Sleator and Tarjan comment that splay trees require no additional storage. We have therefore used a technique that does not call on additional storage for the bottom-up splaying. Top-down splaying never needs to access parent nodes. This study does not compare the storage utilization of the five methods, as all the methods discussed have essentially the same memory requirements.

The trees were compared for a number of tree sizes, activity ratios (ratios of updates to searches) and degree of skewness of key access. We shall present results only for trees of size 4095 nodes ($2^{12} - 1$, to ensure that the trees are of height at least 12) but results for other tree sizes were similar. Four different activity ratios were used. These were 0:100, 20:80, 50:50 and 80:20. The first gives a comparison of the methods in a static environment. The second, i.e. 20 per cent updates to 80 per cent searches, is probably most representative of realistic situations.

To carry out the evaluation, it was necessary to generate probability distributions with prespecified skewness. A common probability distribution that is used for skewed distributions is Zipf's law.[1] Zipf's law specifies that the ith most commonly accessed key out of a total of n possible keys will be accessed with a probability p_i inversely proportional to i. That is,

Table I. BST data structures and operations requiring rotations (A = always; S = sometimes; N = never)

Method	Insert	Delete	Search
Random	N	N	N
AVL	S	S	N
MTR	A	A	A
Splay	A	A	A
Exchange	A	A	A

$$p_i = \frac{C}{i}, \qquad i = 1,\ldots,n$$

where

$$C^{-1} = \sum_{i=1}^{n} \frac{1}{i}$$

Zipf’s law unfortunately does not allow us to generate a number of distributions with various degrees of skewness. It is however possible to generalize Zipf’s law, and a number of such modifications are suggested by Knuth.[1] We present another such modification.

If the probability of access of key k_i is p_i, then p_i is given by

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

where

$$C^{-1} = \sum_{i=1}^{n} \frac{1}{i^{\alpha}} \quad \text{and} \quad \alpha \geq 0$$

For $\alpha = 0$, the probability distribution is uniform, and for $\alpha = 1$ the distribution becomes Zipf’s distribution. For larger values of α, we obtain more highly-skewed probability distributions. Unfortunately, however, it is difficult to relate skewness to the parameter α in the above distribution.

We therefore define another parameter, called the *skew factor*, denoted by β, which is the sum of the probabilities of the most-frequently-accessed 1 per cent of keys. That is,

$$\beta = \sum_{i=1}^{n/100} p_i \, .$$

β gives us the probability of access of the most-frequently-accessed 1 per cent of keys. We have introduced the skew factor, β, as it provides a more intuitive measure than does α of the degree to which the access distribution is skewed. It can be seen from the above equation that β is a function of n and α, although for $\alpha \geq 1$ the dependence of β on n becomes less significant if n is large. For all n, when $\alpha = 0$, we obtain $\beta = 0{\cdot}01$, which indicates that the most-frequently-accessed 1 per cent of the keys are accessed 1 per cent of the time (i.e. the distribution is uniform). For Zipf’s distribution, the value of β grows slowly with n but is approximately 0·5 if n is not small. For $n = 10{,}000$, we obtain $\beta = 0{\cdot}53$. Also, the 80–20 rule (that is, 80 per cent of accesses deal with the most active 20 per cent of the nodes) also leads to a β value of approximately 0·5.

In our evaluation, we have used β values of 0·01, 0·10, 0·20, 0·40, 0·50, 0·60, 0·80 and 0·90 (with corresponding α values of 0, 0·516, 0·687, 0·892, 0·975, 1·058, 1·257 and 1·420, respectively), although only representative results will be presented.

We now describe the evaluation procedure in detail. The evaluation consisted of

1. Building a binary tree of 4095 keys by selecting 4095 unique integer keys randomly from a uniform distribution. No rotations were carried out during this building of the tree except if an AVL tree was being built. The reasons for this are discussed later.
2. Each key inserted in the tree was given a unique randomly-selected position in a table of size 4095. This table will be called the *access probability table*, since the position of the key in this table determines the access probability of that key. The first key in the table is the most-frequently-accessed key and the last key is the least-frequently-accessed. The access probability of the *i*th key in the table is p_i defined using the modified Zipf's distribution discussed above.
3. Carrying out 100,000 insert, delete and search operations in the appropriate ratios specified by the activity ratio. For example, if the activity ratio is $2u$: $100-2u$, then $2u$ updates during each 100 operations were assumed to consist of u deletions and u insertions. To carry out the 100,000 operations, we perform 1000 cycles of 100 operations each. During each cycle the following sequence of operations occurred. Unless there were no updates (i.e. $u=0$), a key k_i from the access-probability table was randomly selected using a uniform distribution. This key k_i was deleted from the tree. Another key (not already in the tree) was then selected uniformly from a large key-space for insertion. This key was inserted as k_i in the access-probability table to replace the deleted key. The process of deleting and inserting keys was repeated u times followed by $100-2u$ search operations using keys selected from the modified Zipf's distribution. The key to be searched was obtained by selecting a random number between 0 and 1 and then finding the index of the corresponding key in the access-probability table using the cumulative probability distribution curve. This completed one cycle. Selecting keys in this manner guaranteed that only successful operations were performed and that the tree size during search was always 4095.
4. Recording the numbers of comparisons and rotations for each type of operation as well as the CPU time during the 100,000 operations. The number of comparisons is assumed to be equal to the number of nodes visited.

It is clear that the above methodology is only one possible model of building, updating and searching a tree to evaluate the performance of the three methods. Many variations of this model are obviously possible, and we considered a number of different models before selecting the present one. We believe that the present choice is a reasonable model of realistic tree activity given that we did not wish to change the access probabilities of the keys in the tree significantly even when some update activity was going on. We wished to keep the access frequencies relatively stable because if the access probabilities changed significantly during the 100,000 operations, we believe that the self-adjusting trees would have performed worse than they did in the present investigation since there would have been significant additional costs due to more drastic restructuring of the tree as a result of dynamically-changing access probabilities.

As noted earlier, when the tree was being built initially, no rotations were performed when a self-adjusting tree was being evaluated. Although the splay tree requires splaying during insertion, we believe no benefit would have been gained as the keys were inserted randomly from a uniform distribution. We considered the possibility of building the initial tree inserting keys such that the keys that were likely to be searched frequently would have had a high probability of getting inserted before those that were searched less frequently. Although we thought this was not realistic, we nevertheless attempted this methodology and found that building trees in this way did not change the results significantly, since the AVL and random trees thus built also had the most-frequently-accessed keys near the root.

During initial building of the tree, rotations were performed in the AVL tree in order to maintain height-balance. Again, to be fair to the self-adjusting trees, we decided to take into account the cost of these rotations by including the cost of building the initial trees in the CPU times that we present.

RESULTS

Table II summarizes the results of the performance evaluation of the six methods when building the initial tree of 4095 nodes was followed by 100,000 operations on the tree in the ratio of 20 per cent updates to 80 per cent searches. The process was repeated for the eight search-key distributions, from uniform to extremely skewed, as listed above.

The table presents average costs of insertions, deletions and searches in terms of numbers of comparisons and rotations as well as the total CPU time for a DECsystem 5100 in megacycles for the 100,000 operations. It should be noted that the number of rotations for the MTR and splay tree are high, since every time a key is updated or searched, a number of rotations are needed to move the key to the root. The average number of rotations for these two methods is therefore almost equal to the average number of comparisons that are needed for each of the operations.

Top-down splaying performs approximately one-fourth the number of rotations that bottom-up splaying and MTR perform. This is due to the fact that: (a) no rotation is performed in the cases when the accessed key is in the left-right or right-left subtree (although splitting involves some small amount of work) and (b) when a rotation is performed we advance two levels down the tree since the root and one of the subtrees is removed. In contrast, the average number of rotations for the exchange method is much smaller, since only one rotation is carried out every time.

We make the following observations about the results in Table II:

1. For a uniform distribution, the exchange method performs poorly. As noted earlier, this performance has been analysed by Allen and Munro,[7] and the mean search cost after one million searches was found to approach the theoretical asymptote cost of $\sqrt{(\pi n)}$ that they have derived. This result can also be seen in Figure 3, which shows how the performance of the exchange method deteriorates with the number of searches under a uniform access distribution. The performance of the other methods does not change significantly with the number of searches.
2. Using the average number of comparisons as a measure of cost (ignoring the not-insignificant cost of rotations, for the moment), the AVL tree clearly has the lowest cost per operation for insertion, and deletion, as well as search when the search-key distribution is uniform. Also, at uniform access distribution, splay, MTR and random trees all have approximately the same mean search

Table II. Mean numbers of comparisons (Comp) and rotations (Rot) and CPU times, for an activity ratio of 20 per cent updates and 80 per cent searches

Skew	Method	Insert		Deletion		Search		CPU
factor	used	Comp	Rot	Comp	Rot	Comp	Rot	time
0·01	Random	15·5	0·0	15·4	0·0	14·7	0·0	24·3
	AVL	12·2	0·6	12·2	0·4	11·2	0·0	24·5
	MTR	15·7	15·7	15·6	13·9	14·8	13·8	104·9
	Exchange	51·9	1·0	51·4	0·9	51·2	1·0	198·0
	BU-splay	17·3	16·3	16·1	14·4	15·4	14·4	138·0
	TD-splay	17·8	4·1	15·8	3·6	15·8	3·6	41·4
0·1	Random	15·5	0·0	15·4	0·0	14·7	0·0	24·4
	AVL	12·2	0·6	12·2	0·4	11·3	0·0	24·6
	MTR	15·7	15·7	15·6	13·9	14·2	13·2	101·6
	Exchange	23·0	1·0	22·7	0·9	20·5	1·0	83·1
	BU-splay	17·3	16·3	16·1	14·5	14·8	13·8	133·7
	TD-splay	17·7	4·1	15·8	3·6	15·2	3·5	40·3
0·4	Random	15·5	0·0	15·4	0·0	14·8	0·0	24·5
	AVL	12·2	0·6	12·2	0·4	11·3	0·0	24·7
	MTR	15·7	15·7	15·5	13·9	11·6	10·6	87·0
	Exchange	19·4	1·0	19·0	0·9	12·6	1·0	55·3
	BU-splay	17·3	16·3	16·1	14·4	12·0	11·0	114·1
	TD-splay	17·7	4·1	15·8	3·6	12·5	2·8	35·2
0·6	Random	15·5	0·0	15·4	0·0	14·9	0·0	24·6
	AVL	12·2	0·6	12·2	0·4	11·3	0·0	24·7
	MTR	15·7	15·7	15·6	13·9	9·6	8·6	76·1
	Exchange	18·7	1·0	18·4	0·9	9·6	0·9	44·7
	BU-splay	17·2	16·2	16·1	14·5	10·0	9·0	99·4
	TD-splay	17·7	4·2	15·8	3·7	10·5	2·3	31·4
0·9	Random	15·5	0·0	15·4	0·0	15·4	0·0	25·1
	AVL	12·2	0·6	12·2	0·4	11·4	0·0	24·8
	MTR	15·6	15·6	15·5	13·9	5·6	4·6	53·8
	Exchange	18·3	1·0	18·0	0·9	4·9	0·8	28·5
	BU-splay	17·2	16·2	16·1	14·4	5·8	4·8	69·6
	TD-splay	17·6	4·1	15·8	3·6	6·3	1·3	23·6

cost in terms of number of comparisons performed, but in terms of CPU time, the randomly-built and AVL trees outperform the self-adjusting trees due to their low maintenance cost.

3. Using the CPU time as a measure of cost, the random tree continues to perform better than the self-adjusting trees even when the skewness is as high as $\beta = 0{\cdot}6$ (that is, 1 per cent of the keys being accessed 60 per cent of the time).
4. If we consider the cost of a rotation carried out during a search of one of the self-adjusting trees to be equal to the cost of one comparison (a rotation would normally be more costly than a comparison), the AVL tree is almost always better if the skewness factor β is less than or equal to 0·6. If we compare the CPU times, the AVL tree is almost 25 per cent faster than the best self-adjusting BST technique (top-down splaying) at this highly-skewed key access.
5. In the case of an extremely-skewed key-access distribution where the skewness

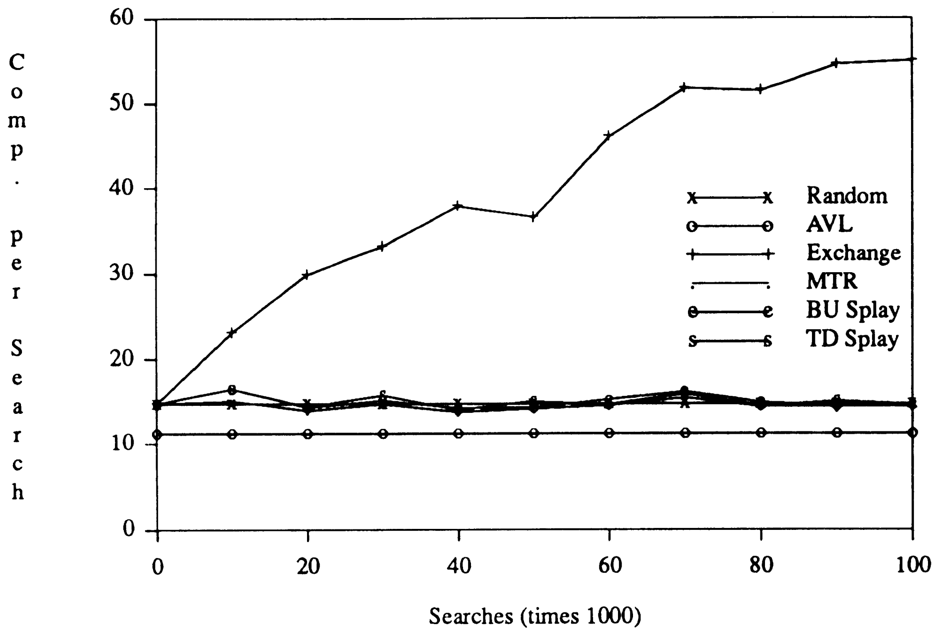

Figure 3. Comparison of asymptotic mean search cost for equiprobable keys

factor is 0·9, the self-adjusting methods all perform well in terms of the average number of comparisons required to access a key. Using average number of comparisons as a measure of cost, the exchange method is best, but if we consider the CPU times, top-down splaying turns out to be the best with the AVL tree not far behind. It should be noted here that the CPU times were obtained on a DECsystem 5100; a RISC machine. The results may vary somewhat on different sequential architectures. We also carried out the evaluation on a CISC architecture machine and found that the ratios of some CPU times were different by as much as 20 per cent, but the basic conclusions were still the same. Results may also vary depending on the compiler technology available, in particular due to the level of optimization performed.

At extremely-skewed access-probability distributions, the AVL tree carries out 2–3 times as many comparisons during search operations as each of the self-adjusting methods, but the performance of the AVL tree in terms of CPU time is still at worst only slightly poorer than that of the self-adjusting trees, in particular top-down splaying.

Note that execution times may be affected by the type of key comparison being performed. A string comparison will generally take much longer than an integer comparison. This may have some effect on relative results. Consider the number of comparisons and CPU times for AVL and TD-splay trees when the skew factor is 0·01 (uniform distribution). Since the TD-splay tree performs more comparisons on average, we might well expect its performance to deteriorate faster than that of the AVL tree if string keys were used. Consider now the situation when the skew factor is 0·90 (highly skewed). We now find that the AVL tree is performing more comparisons than each of the self-adjusting trees. If the cost of a comparison were to increase (due to using string keys), then we might well find the exchange method to be best, since it performs the least comparisons on average.

In Figure 4, we present a summary of mean search costs in terms of number of comparisons for the various methods as the skew factor varies from 0 to 0·9 for an activity ratio of 0:100 (that is, all searches). We also present the theoretical minimum cost of searching a tree with 4095 nodes with search probabilities distributed according to the modified Zipf's distribution for each value of β. This cost was computed as

$$\text{Minimum cost} = \sum_{i=1}^{4095} p_i l_i + 1$$

where l_i is the non-decreasing sequence 0, 1, 1, 2, 2, 2, 2, 3, ... being the level numbers of the nodes of a complete tree of size 4095. It is clear that no tree with 4095 nodes with the given distribution can have a cost lower than this computed minimum cost. Rather than compute the cost for an optimal tree in each case, we decided to use this minimum cost for comparison purposes since the optimal tree was somewhat different every time a tree was built.

Figure 4 shows that the self-adjusting trees do approach the minimum cost as the skew factor approaches 1, whereas (as expected) the costs of AVL and random trees remain relatively unchanged as the skew factor increases. Unfortunately, though, once the cost of rotations is added to the cost of searching in terms of number of comparisons, the self-adjusting trees are not as attractive as they appear in this Figure even at high values of the skew factor.

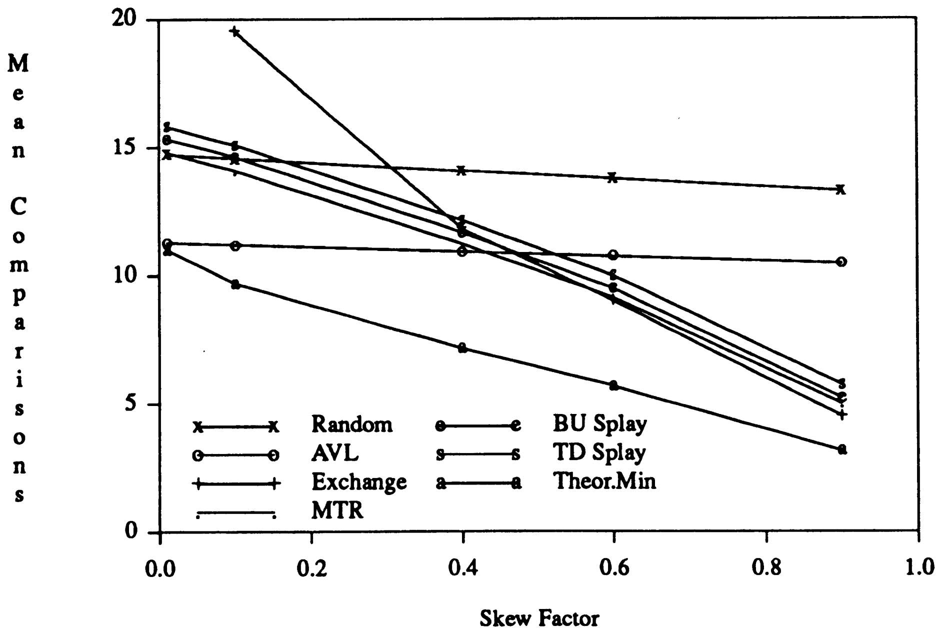

Figure 4. Effect of skew factor on comparisons per search

The effect of changing the activity ratio on the performance of the five methods is shown in Table III. In addition, the performance of the six methods in terms of CPU time needed for 100,000 searches and an activity ratio of 0:100 is displayed graphically in Figure 5. We now make the following observations about the results:

1. When we consider the CPU time as a measure of cost, in all cases where the update ratio is less than 50 per cent, an extremely-skewed distribution is required before any method performs better than AVL.
2. Even when there are no updates and the access probabilities are highly-skewed but fixed (a situation that should be ideal for using self-adjusting trees), the self-adjusting structures perform worse than the AVL tree and are not much better than the random trees.
3. In a highly-dynamic situation where 80 per cent of the operations are updates and only 20 per cent are searches, the random tree performs better than any other method. However top-down splaying performs better than the AVL tree in this case. Since both these methods guarantee logarithmic performance (in an amortized sense) we consider top-down splaying to be preferable in this case. Such highly-dynamic situations are relatively rare, however.
4. Top-down splaying is approximately three times as fast as bottom-up splaying. Sleator and Tarjan[9] presented two different top-down algorithms. One was simpler to code; the other (which we use) was supposedly the faster of the two.

The poor performance of the self-adjusting trees was rather surprising. We note that

Table III. CPU time in megacycles per 100,000 operations. Tree size = 4095 (times include time to build the initial tree)

Activity	Skew	Method used					
ratio	factor	Random	AVL	MTR	Exchange	BU splay	TD splay
0:100	0·01	21·3	15·8	101·6	171·5	135·0	39·4
	0·10	21·1	15·6	96·7	86·0	128·5	37·6
	0·40	20·5	15·4	76·9	53·6	101·9	30·7
	0·60	20·2	15·2	62·5	41·7	82·3	25·6
	0·90	19·6	14·8	34·0	22·3	44·2	15·7
20:80	0·01	24·4	24·6	104·9	198·0	138·1	41·5
	0·10	24·4	24·6	101·6	83·1	133·7	40·3
	0·40	24·5	24·7	87·0	55·3	114·1	35·2
	0·60	24·6	24·7	76·1	44·7	99·4	31·4
	0·90	25·1	24·8	53·8	28·6	69·6	23·6
50:50	0·01	28·7	37·8	109·7	131·7	142·6	44·7
	0·10	28·7	37·8	108·1	76·2	140·5	44·1
	0·40	28·8	37·8	100·3	56·7	129·9	41·3
	0·60	29·0	37·8	93·8	49·8	121·3	39·1
	0·90	29·4	37·8	79·9	38·2	102·8	34·3
80:20	0·01	33·3	51·1	114·5	92·8	147·1	47·9
	0·10	33·3	51·1	114·1	73·4	146·5	47·8
	0·40	33·3	51·1	111·8	58·0	143·5	47·0
	0·60	33·4	51·1	109·5	54·4	140·5	46·2
	0·90	33·5	51·1	104·2	48·1	133·4	44·3

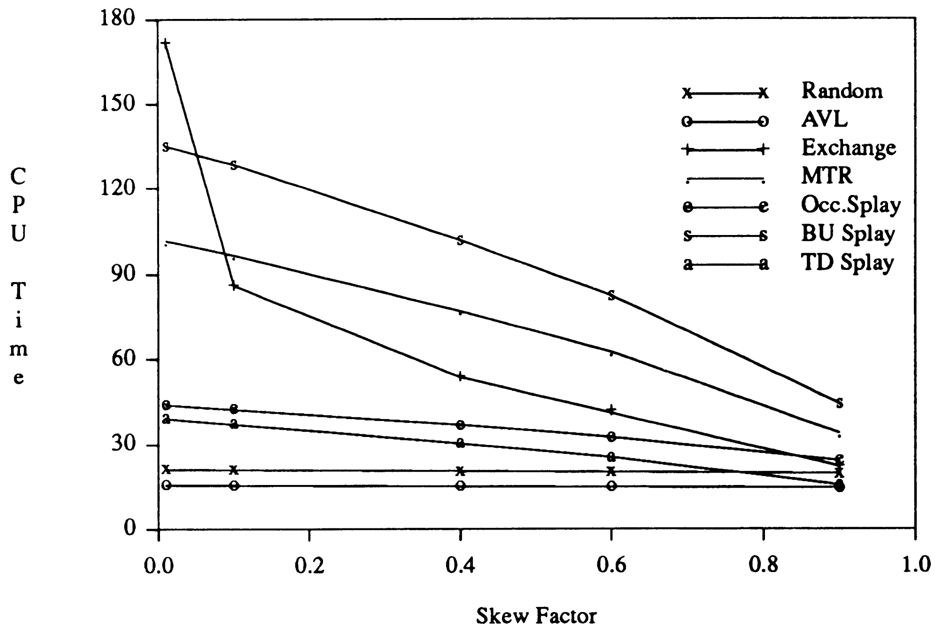

Figure 5. CPU times versus skewness for 100,000 searches

top-down splaying is quite a lot better than other self-adjusting techniques, primarily because it performs fewer rotations and does not require pointer reversal. The primary reason for the poor performance appears to be that the self-adjusting trees perform rotations and accompanying pointer reversals during each search unless the key being accessed is the root. For example, for highly-skewed distributions, say a skewness factor of 0·90, the self-adjusting methods will move all the frequently-accessed keys close to the root, but every time one of these keys is accessed, even if it is close to root, say at level 2, it must be moved to the root. Therefore all the frequently-accessed keys will be moving around continually in the upper levels of the tree. This shuffling represents a waste of time, since promoting one key to the root will in part undo previous work without much saving in the future. This moving to the root has a cost associated with it that almost cancels the benefits that are derived by having the frequently-accessed keys close to the root. An obvious improvement would be to splay only for (say) 10 per cent of key accesses. Another modification to the splay tree (and the other self-adjusting trees) would be to perform no splaying on insertion or deletion. The splay tree that we have evaluated itself is a modification of the splay tree that was proposed by Sleator and Tarjan[9] in that we carried out no splaying on insertions while the initial tree was being built. This was done to improve the performance of the splay tree as compared to the AVL and random trees. The splay tree of Sleator and Tarjan splays on insertion to bring the inserted key to the root and on deletion to bring the predecessor or the successor of the deleted node to the root, assuming that frequently-accessed keys occur in groups. The evaluated implementation of the splay tree carried out splaying after each insertion and deletion that takes place after the initial tree is built. We could further modify the present implementation and eliminate splaying during all insertions and deletions. If deletions/insertions are carried out without splaying then the cost of these operations in a splay tree would be the same as that for a random tree. This modification should help significantly in a dynamic environment where insertions/deletions are occurring frequently.

CONCLUSIONS

We have evaluated the performance of three self-adjusting binary search tree structures and have compared their performance with that of the random BST and the AVL tree. We have found that the search performance of the AVL tree is almost always better than that of any of the self-adjusting BSTs even when the skew factor is close to 1. Of the self-adjusting techniques examined, top-down splaying is easily the best in most situations. Self-adjusting trees perform best in a highly-dynamic environment—which is counter-intuitive. One would expect the best performance to be obtained with a static key set where the trees could adjust properly to the key-access probabilities.

ACKNOWLEDGEMENTS

An earlier version of this paper was presented at the 14th Australian Computer Science Conference held at the University of New South Wales in February 1991. We would like to thank Dr Alistair Moffat of the Department of Computer Science, University of Melbourne for a very careful reading of an earlier version of this paper. His insightful comments have improved this paper considerably.

REFERENCES

1. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addision-Wesley 1973.
2. J. Nievergelt and E. M. Reingold, ‘Binary search trees of bounded balance’, *SIAM J. Computing*, **2**, 33–43 (1973).
3. J. L. Baer, ‘Weight-balanced trees’, *Proceedings AFIPS 1975 NCC*, **44**, 467–472 (1975).
4. H. Chang and S. S. Iyengar, ‘Efficient algorithms to globally balance a binary search tree’, *Comm. ACM*, **27**, 695–702 (1984).
5. Q. F. Stout and B. L. Warren, ‘Tree rebalancing in optimal time and space’, *Comm. ACM*, **29**, 902–908 (1986).
6. E. M. Reingold and W. J. Hansen, *Data Structures*, Little Brown and Company, Boston, 1983.
7. B. Allen and and I. Munro, ‘Self-organising binary search trees’, *JACM*, **25**, 526–535 (1978).
8. J. R. Bitner, ‘Heuristics that dynamically organise data structures’, *SIAM J. Computing*, **8**, 82–110 (1979).
9. D. D. Sleator and R. E. Tarjan, ‘Self-adjusting binary search trees’, *JACM*, **32**, 652–686 (1985).