Course Title

CSE603: Advance Problem Solving

Study & Analysis of Splay Tree, and, Performance Comparison with Red Black Tree

---

Submitted to:

Dr.Venkatesh Choppella

Dr. Vikram Pudi

Submitted By:

Ravi Jakhania -- 2018201018

Bhavi Dhingra -- 2018201058

Project Mentor: Sunchit Sharma

ABSTRACT

The splay tree, a self-adjusting form of binary search tree, is developed and analyzed. Much has been said in praise of self-adjusting data structures, particularly Splay Trees. Splay tree is most suited to skewed key-access distributions as it attempts 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 project, we compare the performance of two different techniques of Splay Tree with that of Red Black Tree. 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 Red Black Tree outperforms Splay Tree in terms of CPU time. Splay tree seem to perform best in a highly dynamic environment, contrary to intuition.

The splay tree, a self-adjusting form of binary search tree, is studied and analyzed. The binary search tree is a data structure for representing tables and lists so that accessing, inserting, and deleting items is easy. On an n-node splay tree, all the standard search tree operations have an amortized time bound of O(log n) per operation, where by “amortized time” is meant the time per operation averaged over a worst-case sequence of operations. Thus splay trees which has two approaches namely, Top down and Bottom up are as efficient as Red Black tree when total running time is the measure of interest. Moreover, Top down approach is a faster than bottom up approach. The efficiency of splay trees comes not from an explicit structural constraint, as with balanced trees, but from applying a simple restructuring heuristic, called splaying, whenever the tree is accessed.

TABLE OF CONTENTS

INTRODUCTION

The Red Black Tree is a commonly-used data structure for storing and retrieving records in main memory because it guarantees logarithmic cost for various operations. 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. The balancing techniques of Red black tree 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, 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, 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 project we consider one of the techniques called splaying. The two different methods of splaying are described in the next section.

The primary aim of this project is to compare the performances of these two methods of splaying with the classical Red-Black tree. The rest of the report is organized as follows. The two methods of splaying to be compared are introduced. This is followed by a description of the methodology used in the comparison of the methods. Finally, we present our results, and the conclusions drawn from these.

SPLAY TREE

The Splay tree structure promotes frequently-accessed keys to the root by modifying the tree at every access, and usually at every insertion and deletion. 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, 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.

Bottom-Up Splaying (BUS)

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. It does the rotations in pairs, in an order that depends on the structure of the access path. To splay a tree at a node x, we repeat the following splaying steps until x is the root of the tree.

Splaying step

Case 1 (zig): If p(x), the parent of x, is the tree root, rotate the edge joining x with p(x).

Case 2 (zig-zig): If p(x) is not the root and x and Ax) are both left or both right children, rotate the edge joining p(x) with its grandparent g(x) and then rotate the edge joining x with p(x).

Case 3 (zig-zag): If p(x) is not the root and x is a left child and p(x) a right child, or vice versa, rotate the edge joining x with Ax) and then rotate the edge joining x with the new p(x)

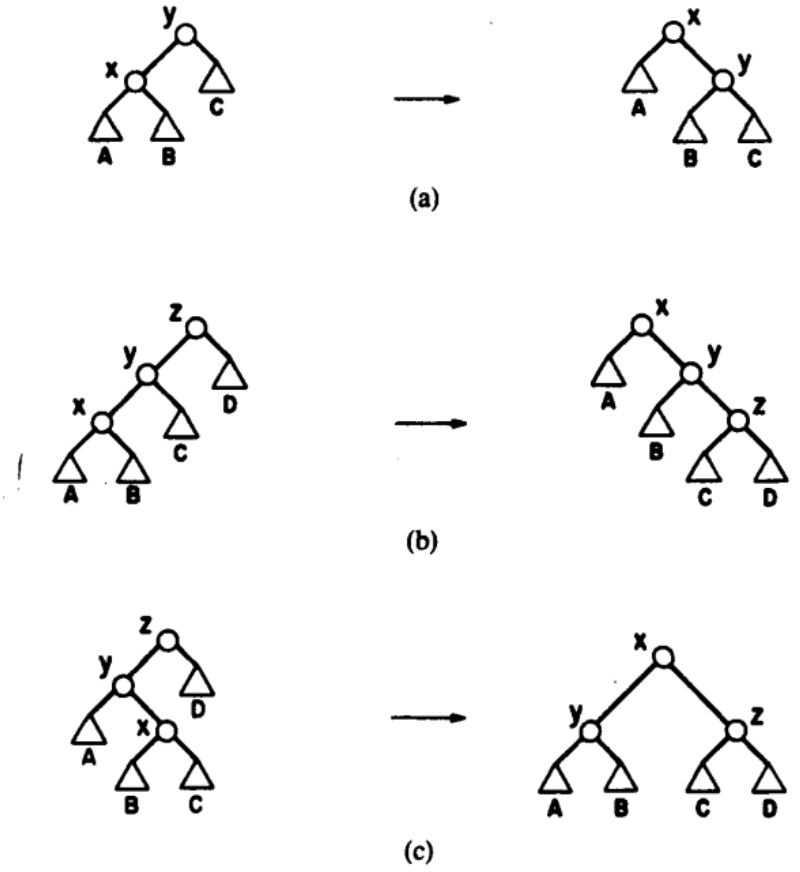

FIG. 1. A splaying step. The node accessed is x. Each case has a symmetric variant.

(a) Zig: terminating single rotation. *(b) Zig-zig:* two single rotations. *(c) Zig-zag:* double rotation.

Top-Down Splaying (TDS)

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.

Top down splay tree has the same amortized performance bounds as Bottom up splay tree; however, no link to parent nodes (either explicit or implicit) is required and fewer rotations are performed. Top down Splay tree performs better than Bottom up Splay tree for basic tree update and search operations. The Red Black trees perform rotations when necessary during insertions and deletions to maintain the constraints. This constraints limits the worst-case search cost. Empirical evidence indicates that the Red Black tree has a mean search cost of O(log(n)) + O(1) under a uniform access distribution.

Table I: BST data structures and operations requiring rotations.

(A = Always; S = Sometimes; N = Never)

Method	Insert	Delete	Search
Red-Black	S	S	N
Splay	A	A	A

AMORTIZED ANALYSIS

To show that splay trees deliver the promised amortized performance, we define a potential function $\Phi(T)$ to keep track of the extra time that can be consumed by later operations on the tree T. As before, we define the **amortized time** taken by a single tree operation that changes the tree from T to T' as the **actual time** t, plus the **change in potential** $\Phi(T')$-$\Phi(T)$. Now consider a sequence of M operations on a tree, taking actual times t_1, t_2, t_3, ..., t_M and producing trees T_1, T_2, ... T_M. The **amortized time** taken by the operations is the sum of the actual times for each operation plus the sum of the changes in potential: $t_1 + t_2 + ... t_M + (\Phi(T_2) - \Phi(T_1)) + (\Phi(T_3) - \Phi(T_2)) + ... + (\Phi(T_M) - \Phi(T_{M-1})) = t_1 + t_2 + ... t_M + \Phi(T_M) - \Phi(T_1)$. Therefore the amortized time for a sequence of operations underestimates the actual time by at most the maximum drop in potential $\Phi(T_M) - \Phi(T_1)$ seen over the whole sequence of operations.

The key to amortized analysis is to define the right potential function. Given a node x in a binary tree, let $size(x)$ be the number of nodes below x (including x). Let $rank(x)$ be the log base 2 of $size(x)$. Then the potential $\Phi(T)$ of a tree T is the sum of the ranks of all of the nodes in the tree. Note that if a tree has n nodes in it, the maximum rank of any node is log n, and therefore the maximum potential of a tree is n log n. This means that over a sequence of operations on the tree, its potential can decrease by at most n log n. So the correction factor to amortized time is at most log n, which is good.

Now, let us consider the amortized time of an operation. The basic operation of splay trees is splaying; it turns out that for a tree t, any splaying operation on a node x takes at most amortized time $3*rank(t) + 1$. Since the rank of the tree is at most log(n), the splaying operation takes O(log n) amortized time. Therefore, the actual time taken by a sequence of n operations on a tree of size n is at most O(log n) per operation.

To obtain the amortized time bound for splaying, we consider each of the possible rotation operations, which take a node x and move it to a new location. We consider that the rotation operation itself takes time t=1. Let $r(x)$ be the rank of x before the rotation, and $r'(x)$ the rank of node x after the rotation. We will show that simple rotation takes amortized time at most $3(r'(x) - r(x))) + 1$, and that the other two rotations take amortized time $3(r'(x) - r(x))$. There can be only one simple rotation (at the top of the tree), so when the amortized time of all the rotations performed during one splaying is added, all the intermediate terms $r(x)$ and $r'(x)$ cancel out and we are left with $3(r(t) - r(x)) + 1$. In the worst case where x is a leaf and has rank 0, this is equal to $3*r(t) + 1$.

Simple rotation

The only two nodes that change rank are x and y. So the cost is $1 + r'(x) - r(x) + r'(y) - r(y)$. Since y decreases in rank, this is at most $1 + r'(x) - r(x)$. Since x increases in rank, $r'(x) - r(x)$ is positive and this is bounded by $1 + 3(r'(x) - r(x))$.

Zig-Zig rotation

Only the nodes x, y, and z change in rank. Since this is a double rotation, we assume it has actual cost 2 and the amortized time is

$$2 + r'(x) - r(x) + r'(y) - r(y) + r'(z) - r(z)$$

Since the new rank of x is the same as the old rank of z, this is equal to

$$2 - r(x) + r'(y) - r(y) + r'(z)$$

The new rank of x is greater than the new rank of y, and the old rank of x is less than the old rank y, so this is at most

$$2 - r(x) + r'(x) - r(x) + r'(z) \quad = \quad 2 + r'(x) - 2r(x) + r'(z)$$

Now, let $s(x)$ be the old size of x and let $s'(x)$ be the new size of x. Consider the term $2r'(x) - r(x) - r'(z)$. This must be at least 2 because it is equal to $\lg(s'(x)/s(x)) + \lg(s'(x)/s'(z))$. Notice that this is the sum of two ratios where $s'(x)$ is on top. Now, because $s'(x) \geq s(x) + s'(z)$, the way to make the sum of the two logarithms as small as possible is to choose $s(x) = s(z) = s'(x)/2$. But in this case the sum of the logs is $1 + 1 = 2$. Therefore the term $2r'(x) - r(x) - r'(z)$ must be at least 2. Substituting it for the red 2 above, we see that the amortized time is at most

$$(2r'(x) - r(x) - r'(z)) + r'(x) - 2r(x) + r'(z) \quad = \quad 3(r'(x) - r(x)) \text{ as required.}$$

Zig-Zag rotation

Again, the amortized time is

$$2 + r'(x) - r(x) + r'(y) - r(y) + r'(z) - r(z)$$

Because the new rank of x is the same as the old rank of z, and the old rank of x is less than the old rank of y, this is

$$2 - r(x) + r'(y) - r(y) + r'(z)$$

$$\leq \quad 2 - 2\, r(x) + r'(y) + r'(z)$$

Now consider the term $2r'(x) - r'(y) - r'(z)$. By the same argument as before, this must be at least 2, so we can replace the constant 2 above while maintaining a bound on amortized time:

$$\leq \quad (2r'(x) - r'(y) - r'(z)) - 2\, r(x) + r'(y) + r'(z) \quad = \quad 2(r'(x) - r(x))$$

Therefore amortized run time in this case too is bounded by $3(r'(x) - r(x))$, and this completes the proof of the amortized complexity of splay tree operations.

COMPARISON METHODOLOGY

Splay trees should not require additional storage than a binary search tree. We have therefore used a technique that does not call on additional storage for the bottom-up splaying while accessing the parent nodes. Top-down splaying never needs to access parent nodes. 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 2^{14} *nodes*, to ensure that the trees are of height at least *14* 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.** Zipf's law specifies that the i^{th} most commonly accessed key out of a total of n possible keys will be accessed with a probability pi inversely proportional to i. That is,

$$p_i = \frac{C}{i} \quad , \quad \text{i = 1,...,n}$$

Where,

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

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. We present another such modification.

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

$$p_i = \frac{C}{i^{\alpha}} \quad , \quad \text{i = 1,...,n}$$

Where,

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

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 percent 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 percent 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 $\beta = 0.01$, which indicates that the most-frequently-accessed 1 percent of the keys are accessed 1 percent 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 $\beta = 0.53$. Also, the 80–20 rule (that is, 80 percent of accesses deal with the most active 20 percent 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 a 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 Splay and Red black tree of 2^{14} keys by selecting 2^{14} unique integer keys randomly from a uniform distribution.
2. Each key inserted in the tree was given a unique randomly-selected position in a table of size 2^{14}. 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 ith key in the table is pi 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 2^{14} **.**

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. 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 Splay tree 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.

RESULTS

Table I summarizes the results of the performance evaluation of the three methods when building the initial tree of 2^{14} 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 the 100,000 operations. It should be noted that the number of rotations for the Splay tree is 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 this method 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. 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.

Table I: *Mean numbers of comparisons (Comp) and rotations (Rot) and CPU times, for an activity ratio of 20 percent updates and 80 percent searches*

Skew Factor	Method used	Insert		Delete		Search		CPU Time (ms)
		Comp	Rot	Comp	Rot	Comp	Rot	
0.01	BU-Splay	43.20	20.35	56.39	26.45	18.47	8.74	17.54
	TD-Splay	41.36	7.48	50.37	10.33	18.68	2.95	13.82
	Red-Black	28.60	0.51	25.68	0.52	25.72	0.00	12.73
0.10	BU-Splay	43.26	20.38	56.48	26.49	18.49	8.74	17.69
	TD-Splay	41.26	7.51	50.65	10.36	18.72	2.94	13.95
	Red-Black	28.61	0.51	25.73	0.52	25.80	0.00	12.96
0.20	BU-Splay	43.23	20.37	56.32	26.41	18.23	8.61	17.03
	TD-Splay	41.39	7.51	50.51	10.31	18.35	2.93	13.48
	Red-Black	28.63	0.51	25.72	0.52	25.85	0.00	12.33
0.40	BU-Splay	43.32	20.41	56.54	26.52	17.36	8.18	15.74
	TD-Splay	41.44	7.53	50.70	10.36	17.60	2.77	12.68
	Red-Black	28.62	0.51	25.71	0.52	25.67	0.00	11.36
0.50	BU-Splay	43.44	20.47	56.72	26.61	16.03	7.52	14.81
	TD-Splay	41.59	7.52	50.89	10.36	16.21	2.57	11.93
	Red-Black	28.68	0.51	25.78	0.52	25.75	0.00	10.93
0.60	BU-Splay	43.52	20.51	56.73	26.62	15.18	7.09	13.77
	TD-Splay	41.67	7.53	50.89	10.36	15.41	2.40	11.25
	Red-Black	28.66	0.51	25.76	0.52	25.90	0.00	10.42
0.8	BU-Splay	43.82	20.66	57.22	26.86	12.01	5.51	10.83
	TD-Splay	42.01	7.55	51.34	10.41	12.10	1.90	9.09
	Red-Black	28.65	0.51	25.74	0.51	25.53	0.00	8.97
0.9	BU-Splay	44.09	20.79	57.51	27.00	9.30	4.15	8.38
	TD-Splay	42.26	7.60	51.57	10.48	9.46	1.42	7.22
	Red-Black	28.67	0.51	25.75	0.52	26.00	0.00	7.90

We make the following observations about the results in *Table I*:

1. Using the average number of comparisons as a measure of cost (ignoring the not-insignificant cost of rotations, for the moment), the Red-Black 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, in terms of CPU time, The Red-Black tree outperforms the Splay trees due to their low maintenance cost.
2. If we compare the CPU times, the Red-Black tree is almost 10 percent faster than the best Splay tree technique (top-down splaying), if the skewness factor β is less than or equal to 0·6.
3. In the case of an extremely-skewed key-access distribution where the skewness factor is 0·9, the Splay Tree methods all perform well in terms of the average number of comparisons required to access a key. If we consider the CPU times, top-down splaying turns out to be the best with the Red-Black tree not far behind. The results may vary somewhat on different systems. 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 Red-Black tree carries out 2–3 times as many comparisons during search operations as each of the Splay Tree methods, but the performance of the Red-Black tree in terms of CPU time is still at worst only slightly poorer than that of the Splay 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 Red-Black 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 Red-Black tree if string keys were used. Consider now the situation when the skew factor is 0·90 (highly skewed). We now find that the Red-Black tree is performing more comparisons than each of the Splay trees.

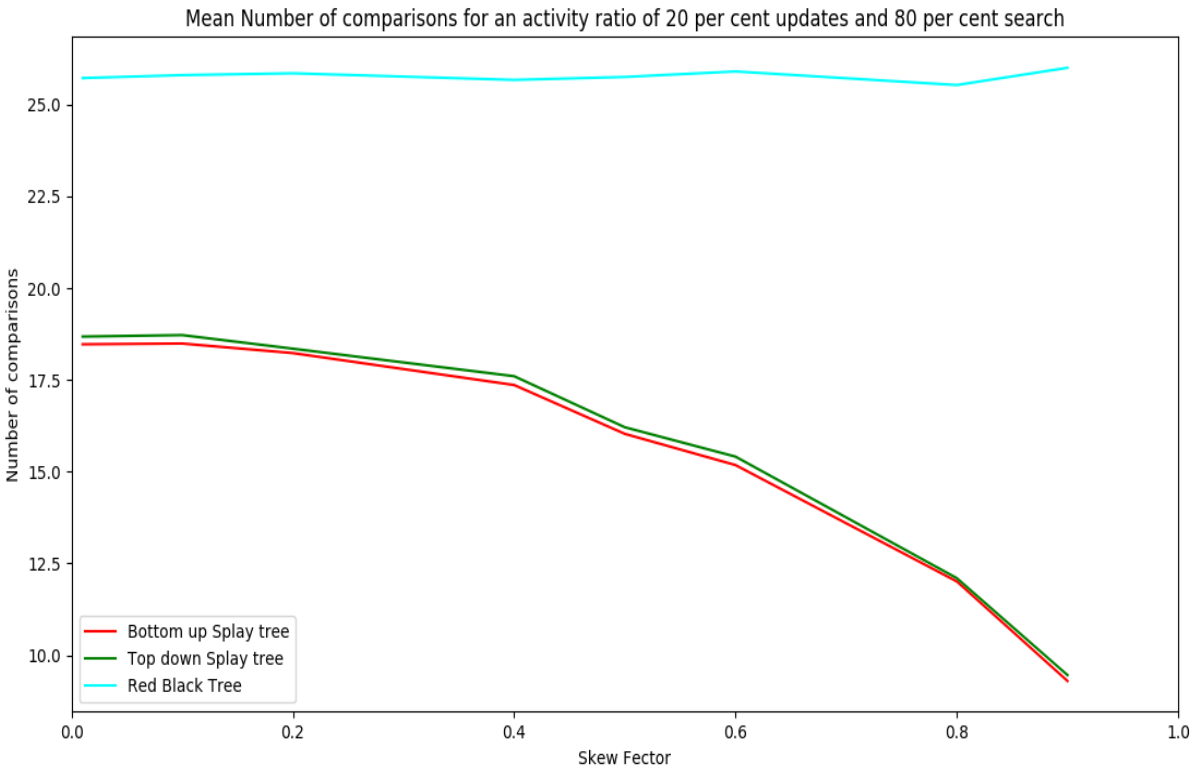

Figure 1

In Figure 1, 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 20:80.

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 Red black tree.
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 Red black tree.
3. In a highly-dynamic situation where 80 per cent of the operations are updates and only 20 per cent are searches, the top-down splaying performs better than the Red black 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.

***Table II*.** CPU time in megacycles per 100,000 operations. Tree size = (4096 * 2 -1) (times include time to build the initial tree)

Activity Ratio	**RBT**	**BUS**	**TDS**
0:100	13.26	15.92	11.36
	13.08	15.91	11.35
	12.50	15.08	10.90
	11.84	14.08	10.30
	10.75	11.94	8.81
	10.36	10.68	7.91
	9.21	8.78	6.67
	8.75	6.91	5.69
20:80	13.04	17.73	13.95
	12.84	17.69	13.98
	12.42	17.19	13.56
	11.50	15.87	12.65
	10.68	14.32	11.46
	9.93	13.15	10.55
	8.71	10.77	8.92
	7.58	7.38	6.38
50:50	9.20	12.30	9.49
	9.04	12.23	9.44
	9.11	12.16	9.37
	8.59	11.27	8.83
	8.33	11.06	8.71
	7.55	9.79	7.74
	6.51	8.25	6.64
	5.93	6.92	5.64

80:20	4.20	5.72	4.22
	3.94	5.46	4.02
	3.96	5.40	3.99
	3.77	5.24	3.90
	3.71	5.22	3.90
	3.61	5.16	3.85
	2.97	3.88	2.95
	2.61	3.49	2.68

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

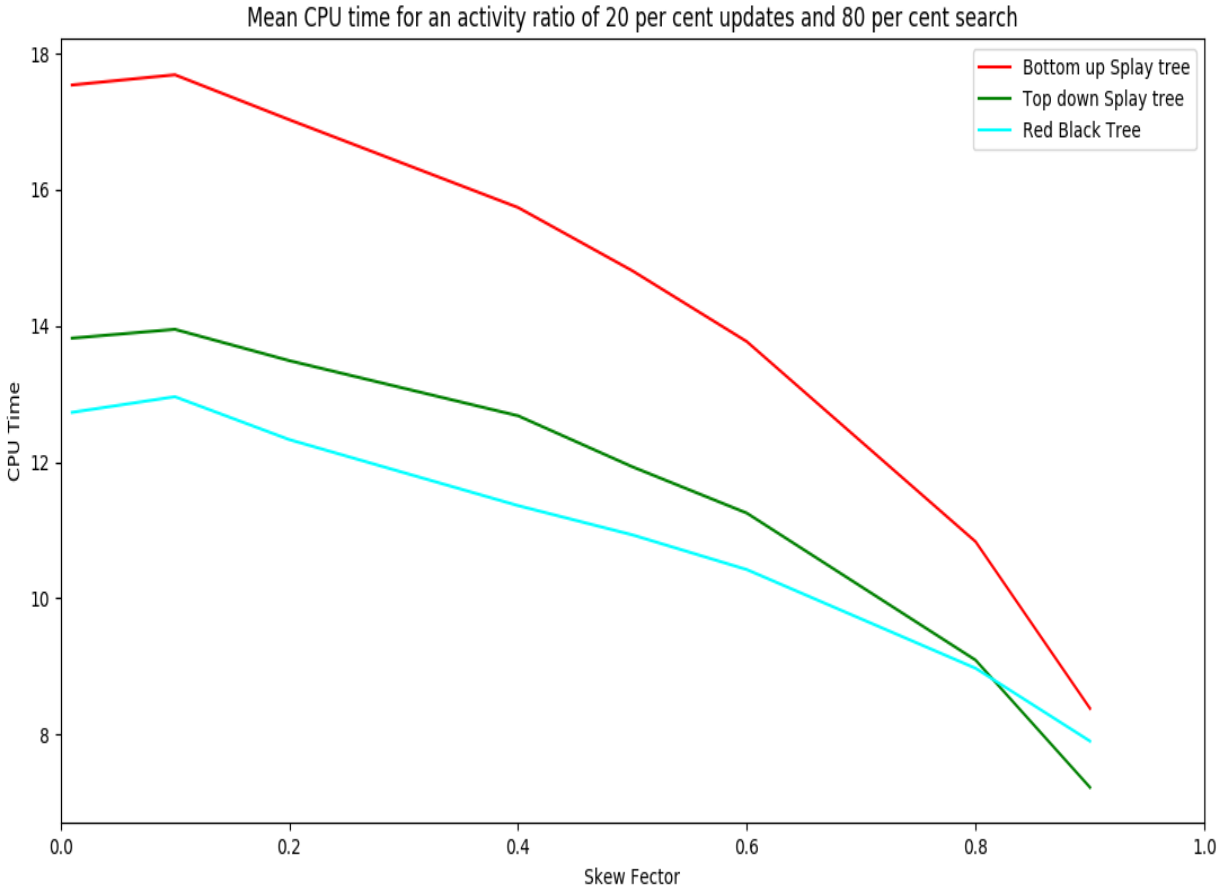

Figure 2

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 that we carried out no splaying on insertions while the initial tree was being built. 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.

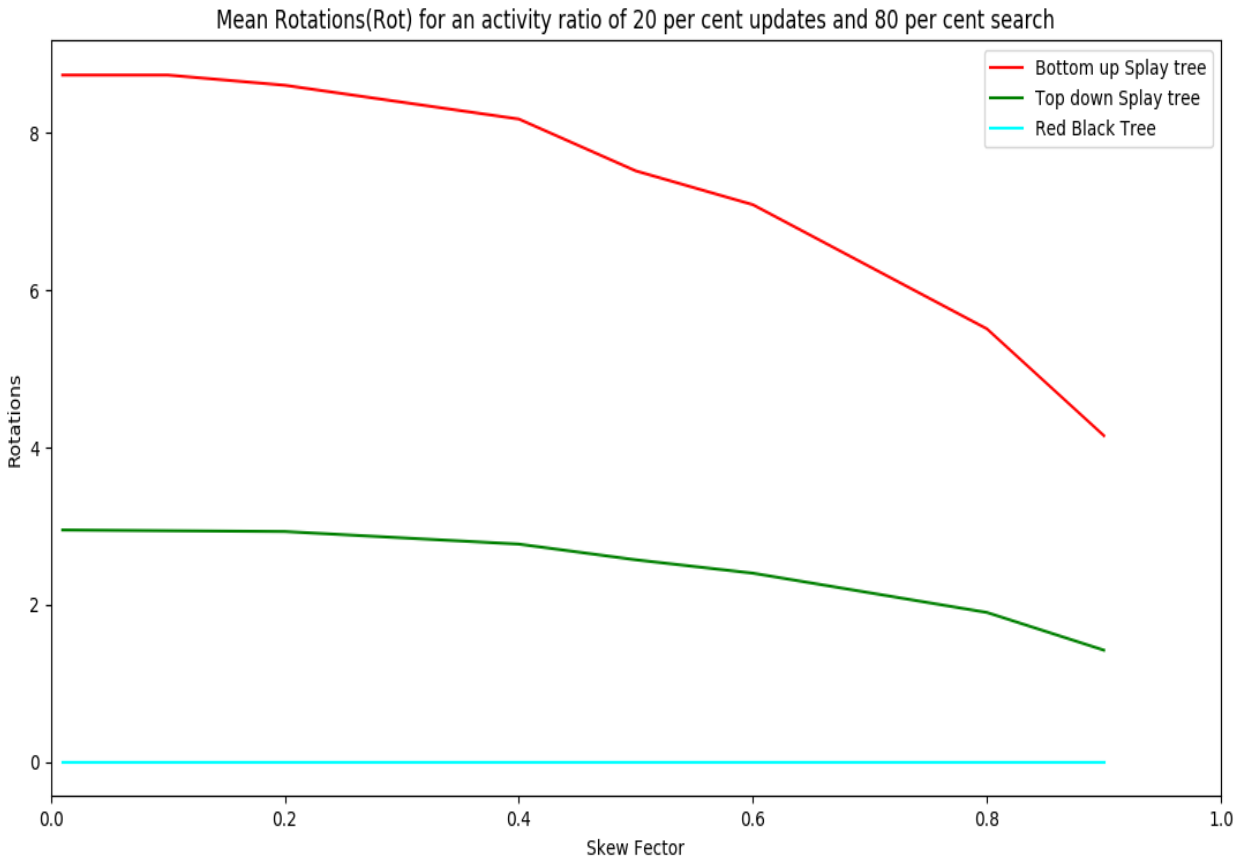
Mean Rotations(Rot) for an activity ratio of 20 per cent updates and 80 per cent search
Bottom up Splay tree
Top down Splay tree
Red Black Tree
Rotations
Skew Fector
0
2
4
6
8
0.0
0.2
0.4
0.6
0.8
1.0

Figure 3

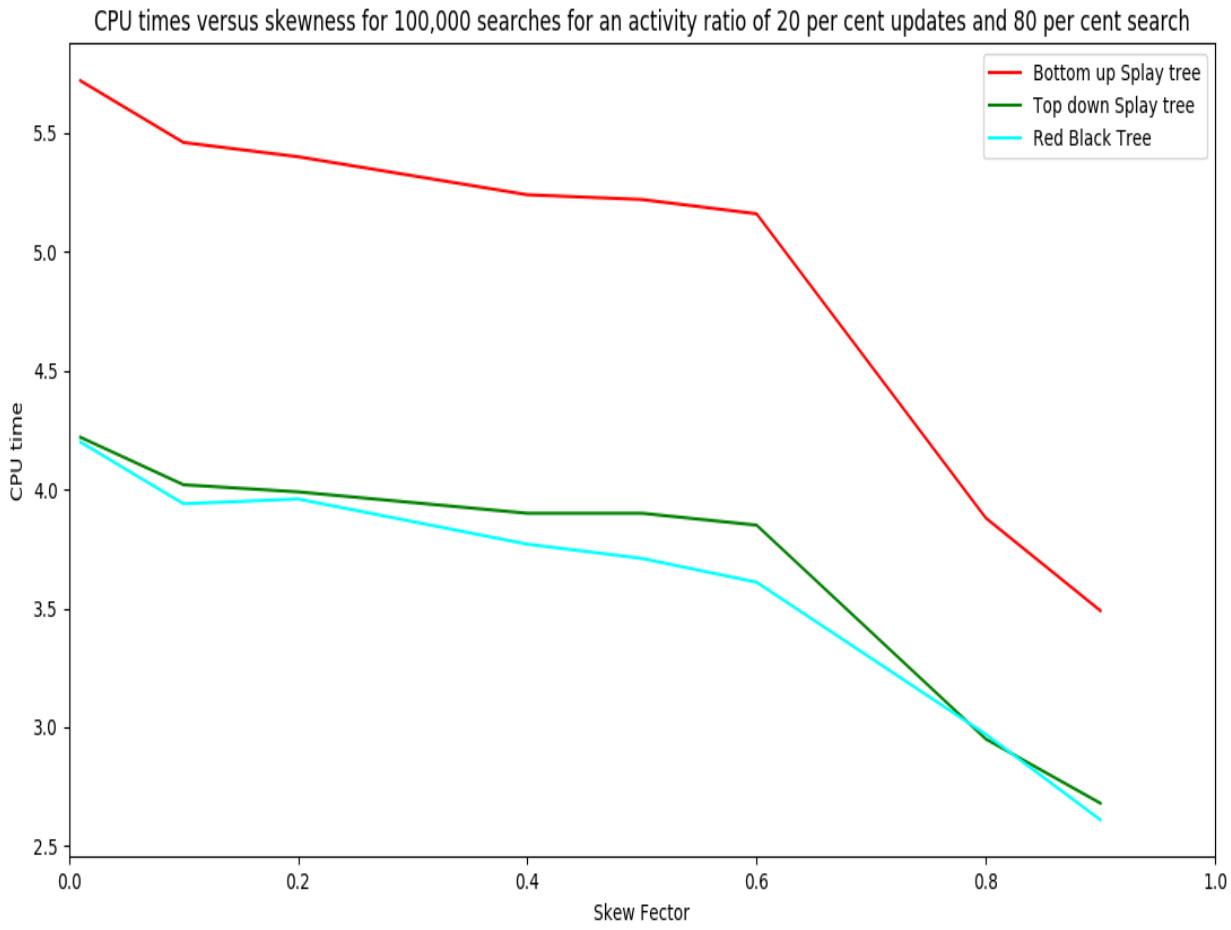

Figure 4

ADVANTAGES OF SPLAY TREE OVER RED BLACK TREE

1. Since Splay tree adjusts according to usage, it can be much more efficient if the usage pattern is skewed.
2. In an amortized sense, ignoring constant factors, Splay tree is never much worse than constrained structures.
3. Splay tree needs less space, since no balance or other constraint information is stored like Red Black Tree
4. Access and update algorithms are conceptually simple and easy to implement.

DISADVANTAGES OF SPLAY TREE OVER RED BLACK TREE

1. Splay tree requires more local adjustments, especially during accesses (look-up operations). While Red Black tree needs adjusting only during updates, not during accesses.
2. Individual operations within a sequence can be expensive, which may be a drawback in real-time applications.

REAL LIFE APPLICATION OF SPLAY TREE

1. Splay tree is used to implement caches. Cache keeps track of the contents of memory locations that were recently requested by processor. It can be made to deliver requests much faster than main memory. Similarly, splay tree uses this concept of accessing the elements quickly that which were recently accessed. Hence, Splay trees are used to implement Cache algorithms.
2. It has the ability of not to store any data, which results in minimization of memory requirements.
3. It can also be used for data compression, e.g. dynamic huffman coding.