Journal of COMPUTER AND FORENSIC SCIENCES

CFS 2022, Vol. 1, Issue 1, pp. 29–45
https://doi.org/10.5937/1-42709
Original research paper

Comparative Performance Evaluation of Suboptimal Binary Search Trees[1]

Svetlana Štrbac-Savić[1], Milo Tomašević[2], Nemanja Maček[3*], Zlatogor Minchev[4]

[1] Academy of Technical and Art Applied Studies, School of Electrical and Computer Engineering, Belgrade, Serbia; svetlana.strbac@viser.edu.rs

[2] School of Electrical Engineering, University of Belgrade, Belgrade, Serbia; mvt@etf.bg.ac.rs

[3] Academy of Technical and Art Applied Studies, School of Electrical and Computer Engineering, Belgrade, Serbia & University Business Academy in Novi Sad, Serbia & SECIT Security Consulting, Serbia; macek.nemanja@gmail.com

[4] Joint Training Simulation and Analysis Center, Institute of ICT, Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, zlatogor@bas.bg

* Corresponding author: macek.nemanja@gmail.com

Received: 2022-11-07 • Accepted: 2022-12-05 • Published: 2022-12-30

Abstract: Three relevant types of suboptimal binary search trees are comparatively evaluated in this paper: two well-known representatives of height-balanced approaches (the AVL and red-black trees) and a popular self-adjusting splay tree. After a brief theoretical background, an evaluation method was described that employs a suitable synthetic workload method capable of producing diverse desired workload characteristics (different distributions and ranges of key values, varying input sequence lengths, etc.). Evaluation analysis was conducted for search, insert, and delete operations separately for each particular type and in appropriate combinations. Experimental results for an average operation cost as well as for tree maintenance cost are comparatively presented and carefully discussed. Finally, the suggested favorable conditions for application of each tree type are summarized.

Keywords: binary search trees; AVL trees; red-black trees; splay-trees; self-adjusting trees.

1. INTRODUCTION

Binary search tree (BST) is a basic data structure that combines two benefits. It allows for fast binary searching of a sorted structure and also, like each dynamic structure, ensures efficient maintenance during the insertion and deletion of keys. It provides the $O(\log n)$ complexity of search, insert, and delete operations in the best and average cases, but for degenerated topologies in the worst case the performance of operations can be deteriorated to $O(n)$. In order to prevent such cases, the tree topology should be kept balanced. However, keeping the optimal balance after each insert or delete operation can impose a

1 This paper in an extension of [1].

Citation: S. Štrbac-Savić, M. Tomašević, N. Maček, and Z. Minchev, "Comparative Performance Evaluation of Suboptimal Binary Search Trees," *Journal of Computer and Forensic Sciences*, 1(1), https://doi.org/10.5937/1-42709.

significant maintenance overhead (up to $O(n)$ sometimes). A compromise is often found in suboptimal BSTs by somewhat relaxing optimal balancing criteria. This approach can significantly reduce the maintenance cost while still guaranteeing the $O(\log n)$ complexity of search even in the worst case with some small constant degradation factor compared to an optimally balanced tree. Different suboptimal strategies have been proposed, but height balancing is the most popular one. The main representatives are AVL trees and red-black trees, which are based on some local sub-tree balancing criteria rather than global ones. In both types of BST (but in different ways), the difference in heights of the leaves is practically restricted within a small constant factor, preventing the linear worst case.

The aforementioned balancing techniques are efficient if the keys in the tree are searched for with nearly uniform probabilities. However, the search probabilities for different keys are often non-uniform, especially when the level of temporal locality is increased. According to that, self-adjusting binary search trees have been proposed that are reorganized even after a search operation. A prominent representative of such an approach is the splay tree, where the successfully found key is moved up to the root in order to exploit the benefits of temporal locality. Since splay trees do not have some explicit balance criteria, the worst case can even go up to $O(n)$, which is acceptable only if it occurs vary rarely. However, the amortized analysis, which gives the time complexity of operations in a series, guarantees $O(\log n)$ complexity in the average case.

The main goal of this paper is to conduct a comparative performance evaluation of AVL, red-black, and splay trees as prominent representatives of suboptimal binary search trees. In order to analyze the performance of these trees under a wide spectrum of different conditions, an appropriate synthetic workload generator is used, which is capable of producing diverse desired workload characteristics. The performance indicators were chosen to be platform- and implementation-independent. The evaluation results should indicate the optimal suggested condition for the employment of these types of trees.

2. MATERIALS AND METHODS

This section provides a brief theoretical background on the AVL, RB, and splay trees, respectively, with their definitions and considerations on the time complexity of the operations.

2.1. AVL Trees

The AVL trees proposed by Adelson-Velski and Landis are height-balanced trees [2]. Let us define the balance of a node as the difference between the heights of its left and right sub-trees. Then, the AVL tree is defined as a binary search tree in which the absolute value of the balance for each node is one at most. The height of an empty tree is defined as 0.

In this way, the balance criterion is considerably relaxed. While the leaves in an optimally balanced tree can be deployed only in two lowest levels, in an extreme case the leaves of the AVL tree can span the range between levels h and $2h$. The worst topology of the AVL tree with maximum height for a given number of nodes is referred to as the Fibonacci tree.

In the AVL trees, the node balance can only be 1, 0, and -1. A node with a balance of 1 leans left, while a node with a balance of -1 leans right. Some insert or delete operations can disturb the balances of the node ancestors on their path to the root, but, as long as they are in the allowed range, there is no need to reorganize the tree. However, when at least one ancestor balance becomes 2 or -2, the specific tree adjusting operations (called single or double rotations) are carried out in order to return the balance of all nodes to the allowed range. The rotations are relatively inexpensive and infrequent, so the overhead of maintaining the AVL tree is quite acceptable [3].

In spite of the fact that the AVL tree is only "nearly" balanced, it was demonstrated in [4] that for an AVL tree with n nodes, its height h satisfies the condition

$$h < 1.4405 \log_2 (n+2) - 0.327 \quad (1)$$

Since the number of comparisons on the search path is determined by the tree height, its finding guarantees that the time complexity of the search is $O(\log n)$, where n denotes the number of nodes in the tree. Along with the same time complexity as in an optimally balanced tree, the degradation factor of the worst case search path is also quite acceptable (less than 45%). Since the eventual rotations in insert and delete operations impose some practically constant additional overhead, the length of the search path is also dominant factor which determines their $O(\log n)$ complexity.

2.2. Red-black Trees

Another nearly balanced topology principally based on height balancing is the red-black tree. While the AVL tree directly restricts the local dis-balance for each node, the red-black tree indirectly controls the length of the search paths by defining the color of each node. It uses an extra bit that denotes a node as red or black and imposes some coloring rules as follows.

The binary search tree is a red-black tree if it satisfies the following conditions [5]:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both his sons are black.
5. Every path from a given node to any of its descendant leaves contains the same number of black nodes.

In the rest of the paper, these trees will be referred to as RB trees.

The RB trees also represent an implementation of the 2-3-4 trees in a form of binary tree [6]. The 2-3-4 trees have optimally balanced topology since all leaves are at the same level. Besides the usual 2-nodes as in the binary trees, these trees may also have 3-nodes with two keys and three sub-trees, as well as 4-nodes with three keys and four sub-trees. The implementation of the 2-3-4 trees requires more memory, and insert and delete operations are more complex since, when necessary, one type of node is transformed into another. Therefore, it is very important that the 2-3-4 trees are B-trees of degree 4, being isomorphic

with the RB tree, which means that for each 2-3-4 tree there exists at least one RB tree as its binary representation [7].

If the tree structure is modified by some insert or delete operation that impairs the requirements of the definition, rotations are performed in order to re-establish the correct structure and coloring. The basic operations of the RB trees have been described in [4, 7] and [8]. It is demonstrated in [5] that the height of the RB tree is bounded by

$$h < 2\log(n+1). \tag{2}$$

Therefore, just like in the AVL trees, the logarithmic performance of all operations if the RB trees is also guaranteed in the average and worst case.

2.3. Splay Trees

Splay trees were proposed by Sleator and Trajan [9]. Although they do not rely on some explicit balancing strategy, unlike AVL and RB trees, the splay trees are reorganized on each access, including even non-invasive search operations, by means of rotations. Each accessed or inserted key is moved to the root, as well as the predecessor/successor of a deleted or unsuccessfully searched key. The rationale behind this is to exploit temporal locality with increased probabilities of accessing recently used keys or range of values.

Two techniques can be employed for tree reorganization: top-down splaying and bottom-up splaying. In bottom-up splaying, the tree is searched for the key in the first step, saving some parent information in nodes on the search path, and then the node is lifted up to the root by consecutive zig, zig-zag, and zag-zag rotations, as described in [10]. In [9], the authors favor top-down playing since it performs in one step without need for an extra storage. Because of that, top-down splaying was used in this evaluation study.

Although splay trees do not guarantee $O(\log n)$ complexity in the worst case, it is demonstrated in [11] that the complexity of performing a series of m operations in splay tree with n keys is

$$O\left(m(1+\log n)+n\log n\right). \tag{3}$$

Consequently, the amortized cost of operations in splay trees is also logarithmic.

3. RELATED WORK

The comparison of different kinds of binary search trees was a goal of many studies. The study from [12] follows a similar approach to our study. Six types of binary search trees were compared: random BST, AVL tree, and four types of self-adjusting binary search trees (splay trees with top-down splaying and bottom-up splaying, and self-adjusting trees with MTR and Exchange techniques described in [13]). It was concluded that AVL trees are the most efficient ones when searching is the most frequent operation, while, among

the self-adjusting structures, the splay trees with top-down splaying technique perform the best in a highly dynamic environment.

In [14], four types of binary search trees were compared: unbalanced BST, AVL tree, RB tree, and splay tree. For each of them, five different node representations were considered: plain, with parent pointers, threaded, right-threaded, and with an in-order linked list. In total, 20 BST variants were compared using three experiments in real-world scenarios with real and artificial workloads. The measured parameters were execution time and the number of comparisons. The results indicate that RB trees are preferred when random input with occasional runs in sorted order is expected. When insertions in sorted order are prevalent, the AVL trees outperform the others for later random access, whereas splay trees perform the best for later sequential or clustered access.

In [15], performance of height-balanced trees (HB[k]) is evaluated. Both analytical and experimental results that show the cost of maintaining HB[k] trees as a function of k are discussed. The AVL tree is treated as a special case for $k = 1$. For the AVL trees, it was concluded that only the search time is a function of the tree size, and in a general case, the maintenance does not depend on the tree size. In general, for HB trees for $k > 1$, the execution times of the procedures for maintaining the HB[k] trees are independent of the tree size, except for the average number of nodes revisited on a delete operation in order to restore the HB[k] property on its trace back. Also, the cost of maintaining HB[k] trees drops significantly as the allowed imbalance (k) increases.

Bear and Schwab in [16] empirically compare the height-balanced trees with the weight-balanced trees by means of simulation with a synthetic workload. In the conclusion, they give preference to the AVL trees.

In [17], a novel limit-splaying heuristic called periodic-rotation is described. It performs splaying after n insertions or accesses in order to reduce the maintenance cost while preserving the performance. They experimentally compared seven data structures: the simple BST, the RB trees, splay trees both with top-down and bottom-up splaying techniques, randomized trees, and their heuristic splay tree. It was presented that such heuristic splay tree where splaying is done periodically rather than on each access is around 27% faster on average than efficient bottom-up splaying. Over five separate text collections that were chosen for workload, several somewhat unexpected conclusions were highlighted: first, top-down splaying is slower than bottom-up splaying in practice; second, bottom-up splaying is about as fast as a self-adjusting randomized tree, but in general is around 25% slower than a BST; and, finally, the most efficient heuristic splaying scheme is only 3% faster than a BST, which performed even better than RB trees.

The study in [18] provides a comparative analysis of a number of different binary search trees: un-optimized BST, AVL tree, several types of the weight-balanced trees (described in [19]), the trees where the searched node moves by one level towards the root [13], as well as the tree with appropriate combinations of some algorithms. The evaluation is based on measured execution times for different types of input sequences. The operations considered were insertion and searching. Although the basic search operation in an ordinary binary tree is quite efficient in many cases, it was concluded that the tree that combines the principles of the AVL tree and an ordinary BST is the most efficient generally.

Although the studies from [12] and [14] are similar to the topic of our study in terms of analyzed trees, the comparisons are carried out from different perspectives. While

some previous studies (e.g., [14]) observed a specific environment in which the trees were applied, our study presents a more general performance study independent of the implementation of the algorithm, operating system, the specific applications, and the machine on which the tests are conducted.

4. EVALUATION METHOD

Although the real workloads are preferred in many studies focused on a specific area of applications, they are unable to reflect a wide spectrum of different workload characteristics. Since an appropriate synthetic workload generator is very convenient for producing diverse desired workload characteristics, our comparative performance evaluation employs the simulation method with specific synthetic workload described in [20]. The main parameters of the workload are: the number of keys, the range of the key values, the distribution of the key values, time locality, the relative frequency of search, insert, and delete operations, the probability of successful and unsuccessful search, etc. The performance indicators have been chosen to be independent of the algorithm implementation and platform on which the measurements are performed (tree height, number of rotations, etc.).

The various key sequences were generated in order to obtain a more complete insight into the chosen trees performance. The intervals from which the key values are taken, the number of elements in the sequence, and the frequency of the key values were varied. A special care was taken to simulate the time locality of the keys in some cases.

The lengths of the key sequences are chosen to be between 10 and 1,000,000 elements in multiples of 10. The number of elements in the sequences was varied in order to establish how the performance depends on the tree size.

The values in the same key sequence may be repeated. They are taken from intervals whose lower bounds are set to zero and whose upper bounds vary from case to case. Four groups of the key sequences used in this evaluation differ according to the way of key generation. The sequences without key repetitions are used for building an initial tree.

The first group of key sequences is sorted in increasing order. They contain unique key values without repeating.

The second group of key sequences is similar to the first, but the order of the key values is random. All values appear exactly once in the sequence, and the length of the key sequence corresponds to the interval upper bound.

In the third group, the keys are also generated randomly. However, the elements are chosen out of a certain interval, whose upper bound also varies, as well as the sequence length. The consequence of such a key choice is that some values from the interval can be repeated, while other values do not appear in the sequences.

The fourth group has the key values that can also be repeated in a sequence, while the sequence lengths and the interval upper bounds are varied like in the third group. However, instead of using random, uniformly distributed key values, the goal was to obtain the key sequences with a non-uniform distribution and to enforce the temporal locality of chosen values, which is sometimes quite pronounced in tree accesses. The function

$$y = \frac{1 - x}{1 + ax} \tag{4}$$

was used to enforce different levels of temporal locality in the following way. First, an initial sequence with n keys without repeated values is formed (let it denote by array *key*). Then, x is randomly chosen from the interval $x \in [0,1]$. With such an x, y is calculated according to equation (3). Since $y \in [0,1]$ for $a > 0$, index i is then calculated as $i = n \cdot y$. Finally, element of the *key* sequence with index i(*key*[i]) is entered into the resulted key sequence. This procedure is repeated until the resulting sequence of the required length is generated.

By varying the parameter a, the shape of the curve can be adjusted, as shown in Figure 1. Values 1, 10, 50, and 100 were taken for parameter a, and the results for a=100 were analyzed. For higher values of a, a uniform distribution of x, values of y are lower. Consequently, lower indices of the key array are much more probable, which increases the time locality in the resulting sequence of key values closer to the start of the array.

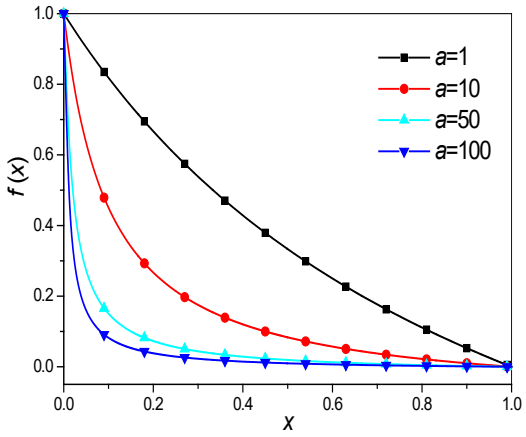

Figure 1. *Function (4) in the interval x ∈ [0, 1] for the different values of parameter a.*

Search, insertion, and deletion operations are evaluated both independently in separate series and jointly in mixed series. In a mixed series, percentages of particular operations are varied, and operations appear randomly according to the adopted frequency.

The following performance indicators were collected during experiments:

- Average height during search operation – average height where the element was found during a successful search or the height of the node where the search was finished in the case of unsuccessful search. It indicates the number of comparisons on the search path.
- Tree height – this parameter refers to the maximum height of the initial tree on which the search series was performed.
- Average number of rotations per operation. It indicates the maintenance cost.
- Tree height at the end of a series of insert operations.
- Average height of the splay tree in an entire series of operations.

5. RESULTS

The experimental results collected from running the series of insert, search, and delete operations are presented and discussed in the first three subsections, respectively. In the last subsection, the results from a series of appropriate combinations of all three operations are shown and analyzed.

5.1. Insert Operation

Since only different values can be inserted in a binary search tree, only the sequences with unique keys are applicable for evaluation of insert operation. The insertions were started with initially empty trees.

Table 1. *Series of insert operations for input key sequences with sorted values in increasing order.*

Number of inserted keys (sorted)	Average height per operation			Average number of rotations per insert operation		Height of resulted tree	
	AVL	RB	Splay	AVL	RB	AVL	RB
100	5.730	8.090	0.990000	0.93000	0.89000	6	10
1,000	8.977	14.481	0.999000	0.99000	0.98300	9	16
10,000	12.362	21.138	0.999900	0.99860	0.99760	13	23
100,000	15.689	27.723	0.999990	0.99983	0.99969	16	30
1,000,000	18.951	34.379	0.999999	0.99998	0.99996	19	36

The results for sorted input key sequences of variable lengths are shown in Table 1. Since the current inserted key is always the highest one, splay trees need no rotations, but after the series of insert operations, the resulting tree has degenerated topology and is far from optimal for most operations that can follow in practice. The AVL trees have a considerably lower average height of the current inserted node, but the RB trees have slightly less average rotations per operation. As a comparison and a rotation are the operations with similar cost, so the AVL tree is more efficient at inserting a sorting sequence. In addition, its final tree height in case of the RB tree is almost twice as tall. Their efficiency is greater considering the final height after the series of insert operations. Since this tree can be the initial tree for some other operations, it can have a serious impact on the cost of the operations that follow.

The results for inserting key sequences generated randomly are given in Table 2. The heights of resulted tree are shown for the AVL and RB tress only since it is very relevant for subsequent search operations, while for splay trees it changes with each operation. Splay trees perform the worst by far in this case, as expected. As for both height indicators, the AVL and RB trees show similar results, while the RB trees have fewer rotations per operation.

Table 2. *Series of insert operations for input key sequences with random values.*

Number of inserted keys (random)	Average height per operation			Average number of rotations per insert operation			Height of resulted tree	
	AVL	RB	Splay	AVL	RB	Splay	AVL	RB
100	5.390	5.410	7.390	0.680	0.550	2.450	7	7
1,000	8.699	8.777	13.929	0.646	0.555	4.954	11	11
10,000	12.056	12.074	20.442	0.645	0.536	7.219	15	15
100,000	15.450	15.523	27.112	0.644	0.534	9.574	19	20
1,000,000	18.815	18.829	33.769	0.640	0.530	11.927	23	23

The results from both Tables 1 and 2 for the same type of tree indicate that the average number of rotations per operation is practically constant over all varied tree sizes for both random and sorted key sequences, except for inserting keys from random sequence in case of splay trees, where this indicator steadily increases with tree size. In the case of most unfavorable sorted input, both AVL and RB trees experience practically a rotation on every insert, while for random input the more efficient RB tree requires a rotation in almost every other insertion. The situation for splay trees is quite opposite, since the maintenance cost for random input is much higher than for a sorted one. Although splay trees most efficiently handle insertion of sorted key sequence, the final tree height has degenerated topology equivalent to a linked list inappropriate for later searching. The fact that the AVL trees have a more restrictive balance criterion contributes to more efficient handling in inserting keys of a sorted sequence, reflected in a considerably smaller average height per operation and final tree height than in the case of the RB trees.

5.2. Delete Operation

A series of delete operations are conducted on initial trees generated with a series of insert operations of random key values in order to be large enough. The heights of the initial trees were 23 for both the AVL and RB trees and 69 for the splay tree. As in the case of insert operations, key sequences with no repeated values were chosen in order to avoid unsuccessful delete operations.

The results presented in Table 3 confirm that splay trees perform the best in cases of deletions of keys from sorted sequences. Each delete operation raises the right subtree in which the next key in sequence is found, making its subsequent deletion more efficient. In the case of splay trees, as the number of operations in a series increases, both the average height of the deleted node and the average number of rotations decrease. Except for a very small percentage of nodes deleted from the initial tree, the RB trees perform better than AVL trees but are still much worse than splay trees.

For the deletion of keys in random order (Table 4), the AVL and RB trees perform very similarly, and their performance indicators only slightly change with the varying number of deleted keys. On the other side, splay trees are again noticeably less efficient, and their performance deteriorates with an increasing number of deleted keys in random order.

Table 3. *Series of delete operations on key sequences with sorted values in increasing order.*

Number of deleted keys (sorted)	Average height of deleted node per operation			Average number of rotations per delete operation		
	AVL	RB	Splay	AVL	RB	Splay
100	15.520	15.980	2.210	0.600	0.700	0.890
1,000	14.710	15.793	1.500	0.572	0.680	0.546
10,000	14.730	14.717	1.392	0.578	0.662	0.497
100,000	14.150	14.083	1.361	0.578	0.659	0.482
1,000,000	13.708	12.534	1.347	0.575	0.658	0.475

Comparing the results for the sorted and random order of deleted keys in splay trees, two opposite trends can be noticed. Longer sorted key sequences during deletions are favorable, while longer random ones are unfavorable for both the average height of the deleted node and the average number of rotations. The AVL and RB trees have a larger average height and a smaller average number of rotations for the same number of delete operations in the case of a random key sequence. Also, unlike splay trees, these performance indicators for AVL and RB trees are relatively insensitive to the number of deleted keys.

Table 4. *Series of delete operations on key sequences in random order.*

Number of deleted keys (random)	Average height of deleted node per operation			Average number of rotations per delete operation		
	AVL	RB	Splay	AVL	RB	Splay
100	15.060	16.000	8.550	0.330	0.320	3.160
1,000	15.835	16.321	14.648	0.384	0.357	5.458
10,000	15.402	16.541	21.745	0.351	0.359	8.166
100,000	16.703	16.865	28.671	0.357	0.361	10.747
1,000,000	16.465	16.437	35.687	0.378	0.387	13.359

5.3. Search Operation

The search operation is especially important because it is usually the most frequent operation and also because it is the first part of insert and delete operations. This is the reason why the performance of this operation is analyzed in more detail.

Search operations in the AVL and RB trees do not modify the tree topology, and no rotations are required. Therefore, the average length of the search path is the only relevant performance indicator. However, in splay trees, every search operation is followed by an adjustment of the topology, and the average number of rotations is meaningful as well. All the results presented in Table 5 are obtained by searching for a sorted sequence of keys in increasing order. The heights of the initial trees were 15 for the AVL and RB trees and 43 for the splay tree. In the first three cases (up to 10,000 keys searched), all searches were successful, while in the other two cases, there were 90% and 99% unsuccessful searches.

Table 5. *Series of search operations on key sequences with sorted values in increasing order.*

Number of searched keys (sorted)	Average search path length per operation			Average number of rotations (splay tree)
	AVL	RB	Splay	
100	12.260	11.160	2.790	0.670
1,000	11.697	11.472	2.460	0.515
10,000	11.578	11.564	2.348	0.473
100,000	14.656	12.858	1.135	0.047
1,000,000	14.966	12.986	1.013	0.048

Again, the splay trees are obviously the most efficient ones when the key sequence is sorted. With longer search sequences, their performance is steadily improving. After the first operation in a series radically rearranges the tree topology, each subsequent search slightly adjusts it to make the subsequent operation more efficient. Finally, when unsuccessful search operations for key values higher than the maximum key in the tree prevail, they execute very fast since no further tree adjustments are needed. However, the average cost of search operations in the AVL and RB trees depends greatly on the initial tree size since there are no adjustments during the series of search operations. Unlike splay trees, in case of AVL and RB trees performance is deteriorated when the number of keys in the sorting sequence grows due to prevailing number of unsuccessful search operations (their search paths are ended in leaves of the tree). The performance of the AVL tree is especially affected in this case.

Table 6. *Series of 100,000 search operations on key sequences with random distribution.*

Range of searched key values (random)	Average search path length per operation			Average number of rotations (splay tree)
	AVL	RB	Splay	
0..99	13.741	14.682	6.254	2.059
0..999	13.299	14.256	10.868	3.574
0..9,999	13.756	14.813	15.662	5.111
0..999,999	16.743	15.853	3.421	0.884

Table 6 presents the results for sequences of 100,000 search operations with random distribution of key values performed on initial trees which were built with series of insert operations of random keys values between 0 and 99,999. Table 7 shows the results obtained under the same conditions but with a non-uniform distribution and enforced temporal locality of the searched key values. The heights of the initial trees were 19 for the AVL tree, 20 for the RB tree, and 56 for the splay tree. In cases when range of key values searched was 0..999,999, there were approximately 90% unsuccessful search operations, while in other sequences all searches were successful.

Splay trees clearly outperform the others when non-uniform sequences of key values are searched since they can take advantage of increased temporal locality (much better indicators for splay trees in Table 7 compared to those in Table 6) by proper adjustment of the topology, while the AVL and RB trees are insensitive to this phenomenon. Their performance rather depends on a set of key values and their place in the initial tree. For both distributions, the average cost of a successful search operation in the AVL trees is slightly better than in the RB tree. The large percentage of unsuccessful operations affected the performance of the analyzed trees in the same manner as in the previous case.

Table 7. *Series of 100,000 search operations on key sequences with non-uniform distribution.*

Range of searched key values (non-uniform)	Average search path length per operation			Average number of rotations (splay tree)
	AVL	RB	Splay	
0..99	14.043	13.283	1.916	0.593
0..999	13.307	14.183	5.829	1.919
0..9,999	13.290	14.374	10.296	3.406
0..999,999	16.712	15.820	2.763	0.666

Figure 2 depicts how the distribution of key values in the search sequences affects the average length of the search path. It more explicitly demonstrates much better handling of increased temporal locality of search sequences in splay trees than in the AVL and RB trees.

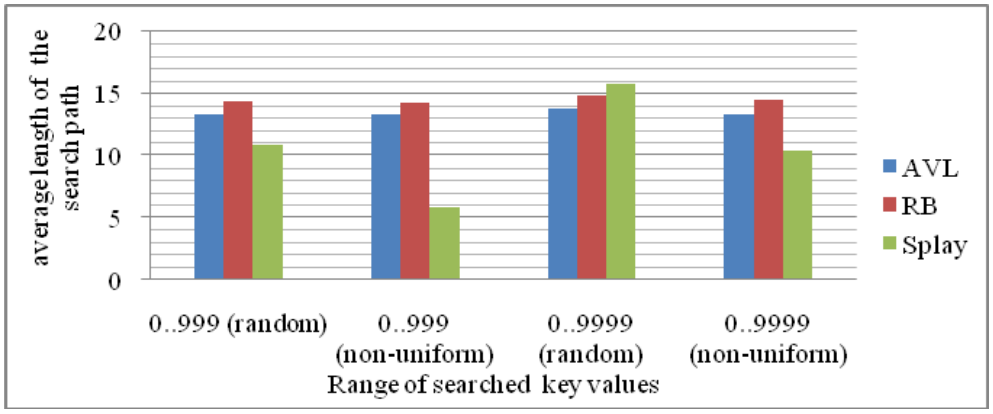

Figure 2. *Average lengths of search path for series of search operations for random and non-uniform key sequences.*

In previous experiments, unsuccessful search operations for key values in the interval 100,000..999,999 traversed the right-most search part, which is not quite typical. Therefore, a more realistic situation with unsuccessful searches dispersed across an entire tree should be simulated. To this end, about 20,000 randomly chosen key values were deleted from an initial tree randomly built with 100,000 keys. The obtained tree was used for the evaluation of search sequences of 100,000 keys. Four ranges of key values (0..29,999, 0..49,999, 0..79,999, 0..99,999) and two distributions (random and non-uniform) are varied to produce six new sequences.

The results from such input sequences are given in Table 8 for random distribution and in Table 9 for non-uniform distribution. The efficiency of search operations in the AVL and RB trees practically does not depend on the range of search keys or type of distribution. A number of unsuccessful search operations increases their average search path to some extent. The AVL tree is again slightly better than the RB tree, but both types of trees outperform the splay tree, especially for randomly distributed searched key values. Although enforced temporal locality in non-uniform distribution evidently improves the performance of search operations and the maintenance cost of the splay tree, it is not sufficient to make it better than the AVL and RB trees in conditions when unsuccessful search operations are spread over a larger range of key values.

Table 8. *Series of 100,000 search operations in different ranges of searched key values (random distribution).*

Range of searched key values (random)	Average search path length per operation			Average number of rotations (splay tree)
	AVL	RB	Splay	
0..99,999	14.991	15.062	22.208	7.188
0..79,999	14.903	15.224	20.503	6.672
0..49,999	14.810	15.573	19.569	6.370
0..29,999	14.557	15.566	18.249	5.940

Table 9. *Series of 100,000 search operations in different ranges of searched key values (non-uniform distribution).*

Range of searched key values (non-uniform)	Average search path length per operation			Average number of rotations (splay tree)
	AVL	RB	Splay	
0..99,999	14.448	14.426	16.829	5.502
0..79,999	15.152	15.479	16.172	5.290
0..49,999	15.365	16.075	16.536	5.406
0..29,999	14.675	15.656	15.797	5.027

5. DISCUSSION

Finally, after insert, delete, and search operations are analyzed separately, a more realistic situation when different operations are interspersed is in place. Different workload characteristics are also simulated by varying the relative frequencies of these three types in a sequence. Two different series of operations are analyzed. It was assumed that the search operation is the most frequent one, while delete and insert operations are equally represented in this evaluation. The initial tree was built from the values in the interval [0, 99999] inserted in random order. The cost of building the initial tree is not accounted for in the evaluation of the mixed series, which has 100,000 operations.

Table 10. *Series of 80% search, 10% insert, and 10% delete operations with random distribution of key values.*

Range of key values (random)	Average height per operation			Average number of rotations per operation		
	AVL	RB	Splay	AVL	RB	Splay
0..9	14,189	15,897	1,290	0,021	0,015	0,143
0..99	13,325	15,813	2,157	0,023	0,018	0,463
0..999	11,623	13,209	7,120	0,023	0,020	2,319
0..9,999	14,356	15,359	16,556	0,029	0,028	5,499
0..99,999	14,711	14,782	21,957	0,033	0,030	7,309
0..999,999	17,925	18,017	19,020	0,066	0,055	6,275

The results for the first mixed series made of 80% search, 10% insert, and 10% delete operations with the key values from different ranges in random order are presented in

Table 10. Splay trees are mostly sensitive to the range of key values. For smaller ranges, they have a smaller average height per operation than both AVL and RB trees at the expense of rotations constantly used to adjust the tree. However, as the growing range of values decreases, the temporal locality and its average performance become significantly worse. When compared to the RB trees, the AVL trees, as a more restrictive topology, require slightly more rotations, but it pays off in a smaller average height per operation due to the prevalent number of search operations. For smaller ranges of key values, both the AVL and RB trees are less sensitive to this parameter. However, a very large range of key values impairs their performance because of the increased incidence of unsuccessful search operations ending in the leaves.

Table 11 shows the experimental results for the same frequencies of operations as before, but the key values in mixed sequences follow the non-uniform distribution. It is evident again that splay trees perform noticeably better for key values with a non-uniform distribution of exploiting enforced temporal locality. On the other side, there is no consistent effect on the performance of the height-balanced representatives for this distribution. The AVL trees are still slightly better.

Table 11. *Series of 80% search, 10% insert, and 10% delete operations with non-uniform distribution of key values.*

Range of key values (non-uniform)	Average height per operation			Average number of rotations per operation		
	AVL	RB	Splay	AVL	RB	Splay
0..9	15,057	15,127	0,987	0,023	0,019	0,031
0..99	15,025	15,176	2,809	0,018	0,010	0,630
0..999	14,841	15,899	8,336	0,009	0,006	2,775
0..9,999	15,395	16,447	13,440	0,014	0,013	4,482
0..99,999	15,265	15,292	18,195	0,023	0,021	6,071
0..999,999	16,961	17,168	17,345	0,060	0,050	5,748

Table 12. *Series of 50% search, 25% insert, and 25% delete operations with random distribution of key values.*

Range of key values (random)	Average height per operation			Average number of rotations per operation		
	AVL	RB	Splay	AVL	RB	Splay
0..9	14,459	15,729	1,724	0,052	0,040	0,357
0..99	14,231	15,594	4,058	0,056	0,046	0,120
0..999	12,716	13,618	7,402	0,058	0,048	2,395
0..9,999	14,174	15,168	16,869	0,066	0,059	5,694
0..99,999	14,795	14,864	23,330	0,083	0,073	7,970
0..999,999	17,101	17,660	21,158	0,165	0,136	7,071

Table 13. *Series of 50% search, 25% insert, and 25% delete operations with non-uniform distribution of key values.*

Range of key values (non-uniform)	Average height per operation			Average number of rotations per operation		
	AVL	RB	Splay	AVL	RB	Splay
0..9	15,139	15,289	0,948	0,058	0,047	0,065
0..99	14,682	15,046	2,924	0,042	0,035	0,779
0..999	14,686	15,794	8,193	0,018	0,014	2,773
0..9,999	15,323	16,432	13,526	0,031	0,028	4,584
0..99,999	15,382	15,409	18,881	0,042	0,037	6,423
0..999,999	16,416	16,836	18,810	0,130	0,109	6,353

After that, the relative frequencies for the second mixed series were set to 50% for search, 25% for insert, and 25% for delete operations to see how a higher percentage of input and delete operations affected the average cost per operation. The results for sequences with key values from different ranges are presented in Table 12 for random distributions and in Table 13 for non-uniform distribution. In case of the AVL and RB trees, the average number of rotations is growing almost linearly with the increased percentage of insert and delete operations. The cost of tree maintenance in splay trees is not much affected since they adjust the tree topology on each access. It seems that different relative frequencies of three operations different do not have significant impact on an average height per operation in all trees for both random and non-uniform distributions. Suitability of non-uniform distribution and smaller ranges of key values for splay trees is evidenced once again.

6. CONCLUSIONS

It can be concluded that the AVL and red-black trees perform quite similarly. However, in a number of cases, the AVL trees are slightly more efficient than their red-black counterparts in terms of average height per operation, especially in sequences of search operations and in sequences of mixed operations, as a consequence of their more stringent topology requirements. It comes at the expense of somewhat increased maintenance costs expressed in an average number of rotations. The red-black trees are more efficient when the key values in sequences are unique and random. Both types of trees are rather insensitive to the order of key values and temporal locality. On the other hand, the splay trees prefer situations where key values come in sorted order. They especially outperform the others when the temporal locality of accesses is increased and when searching for a rather narrow range of the key values. In these cases, the high cost of constant tree maintenance is amortized; otherwise, it can be intolerable. This conclusion indicates that a modified splay tree could be proposed that dynamically tracks the temporal locality of key values (e.g., access counters) and adjusts the tree topology only when it is justified.

For more information on AVL trees, readers may consult [15]. Details on the AVL-based settlement algorithm and reservation system for smart parking systems in IoT-based smart cities are presented in [16]. More details on red-black trees can be found in [17]. To read more on splay trees, readers may cf. [18]. To learn more about augmented binary search trees, cf. [19].

FUNDING:

This research received no external funding.

CONFLICTS OF INTEREST:

The authors declare no conflict of interest.

REFERENCES

[1] S. Štrbac-Savić and M. Tomašević, "Comparative performance evaluation of the AVL and red-black trees," In Proceedings of the Fifth Balkan Conference in Informatics, September (BCI '12), 2012, pp. 14–19.

[2] G. M. Adelson-Velskii and E. M. Landis, "An Algorithm for the Organization of Information," *Soviet Mathematics Doklady*, vol. 3, pp. 1259–1263, 1962.

[3] M. Tomašević, *Algorithms and Data Structures*. Belgrade, Serbia: Academic Mind, (in Serbian), 2011.

[4] D.E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Reading, Massachusetts: Addison-Wesley, 1998.

[5] T. Cormen, Ch. Leiserson, and R. Rivest, *Introduction to Algorithms*. The MIT Press, McGraw-Hill, 2009.

[6] A. Aho, J. Hopcroft, and J. Ullman, *Data Structures and Algorithms*. Addison-Wesley, 1983.

[7] B. Flaming, *Practical Data Structures in C++*. New York: John Wiley and Sons, 1993.

[8] C. Okasaki, "Red-black trees in a functional setting," *Journal of Functional Programming*, Vol. 9, No. 4, pp. 471–477, 1999.

[9] D. Sleator and R. E. Trajan, "Self-Adjusting Binary Search Trees," *Journal of the Association for Computing Machinery*, Vol. 32, No. 3, pp. 652–686, 1985.

[10] M. A. Weiss, *Data Structures and Algorithm Analysis in C*. Reading, MA: Addison-Wesley, 1997.

[11] R. Cole, "On the Dynamic Finger Conjecture for Splay Trees, Part II: The Proof," *SIAM Journal on Computing*, Vol. 30, pp. 44–85, 2000.

[12] J. Bell and G. Gupta, "An Evaluation of Self-Adjusting Binary Search Tree Techniques," *Software – Practice and Experience*, Vol. 23, pp. 369–382, 1993.

[13] B. Allen and I. Munro, "Self-Organising Binary Search Trees," *JACM*, Vol. 25, pp. 526–535, 1978.

[14] B. Pfaff, "Performance Analysis of BSTs in System Software," *ACM SIGMETRICS*, Vol. 32, Issue 1, pp. 410–411, 2004.

[15] R. Wiener, "AVL Trees," In *Generic Data Structures and Algorithms in Go*, Berkeley, CA: Apress, 2022, pp. 315–347.

[16] H. Canli and S. Toklu, "AVL Based Settlement Algorithm and Reservation System for Smart Parking Systems in IoT-based Smart Cities," *The International Arab Journal of Information Technology*, Vol. 19, No. 5, pp. 793–801, 2022.

[17] R. Wiener, "Red-Black Trees," In *Generic Data Structures and Algorithms in Go*, Berkeley, CA: Apress, 2022, pp. 363–385.

[18] B. A. Berendsohn and L. Kozma, "Splay trees on trees," In Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2022, pp. 1875–1900.

[19] T. Luo, "Learning Augmented Binary Search Trees," Doctoral thesis, Carnegie Mellon University, Pittsburgh, PA, 2022.