Comparative Performance Evaluation of the AVL and Red-Black Trees

Svetlana Štrbac-Savić
School of Electrical Engineering and Computer Science Applied Studies
Vojvode Stepe 283, Belgrade, Serbia
+381-11-3950026
svetlanas@viser.edu.rs

Milo Tomašević
School of Electrical Engineering
Bulevar Kralja Aleksandra 73, Belgrade, Serbia
+381-11-3218392
mvt@etf.rs

ABSTRACT

The AVL and red-black trees are the suboptimal variants of the binary search trees which can achieve the logarithmic performance of the search operation withot an excessive cost of the optimal balancing. After presenting a brief theoretical background, the paper comparatively evaluates the performance of these two structures. The evaluation was performed by means of simulation with a synthetic workload model. In order to obtain a better insight, the performance indicators are chosen to be implementation and platform independent. Some representative results of the evalution are given and discussed. Finally, the findings of this study are summarized into the suggestions for an optimal use of the analyzed trees.

Categories and Subject Descriptors

E.1 [**Data Structures**]: *Trees*; F2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems – *Sorting and searching*

General Terms

Algorithms, Performance

Keywords

Binary search trees, AVL trees, red-black trees

1. INTRODUCTION

The binary search tree (BST) represents one of the basic structures intended for an efficient searching and maintaining the dynamic data sets in the main memory [1]. An advantage of the binary search tree comes from their dynamic structure which ensures the efficient insertions and deletions. A more important characteristic is that BST guarantees the logarithmic complexity of the search, insert, and delete operations in the best and average case. The techniques of the tree balancing are essential in order to avoid the worst case topology and to achieve the optimum search performance. However, since the perfect balancing after each node insertion and deletion may be very expensive (even up to $O(log\ n)$ in the worst case), it is often resorted to suboptimum variants of this tree. They relax the balancing criteria and the balancing overhead dramatically decreases. The logarithmic order of the search operation complexity, the same as in the optimally balanced structure, is still guaranteed with a quite acceptable constant degradation factor.

The different structures have been proposed, based on the different balancing techniques (e.g., the weight balancing in the bounded balanced trees), but the most popular and the most widely used BST structures are based on the height balancing. The most representative structures of this type are the AVL trees and the red black trees. The main goal of this paper is to perform a qualitative comparative performance evaluation of these two structures by using a suitably chosen synthetic workload model. In the second section, the definition of AVL tree is given and some characteristics are discussed, while the third section elaborates on the same issues for the red black trees. In the fourth section, the evaluation method and the workload model are described. In the fifth section, some quantative results of the comparative evaluation are presented and discussed.

2. THE AVL TREES

Let us define the *balance* of a node *b* as the difference between the heights of its left and right subtree (*hl* and *hr,* respectively). 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 the most [2]. The height of an empty tree is defined as 0.

In this way, the balance criterion is considerably relaxed. For example, unlike the optimally balanced tree where the leaves can be deployed only in two lowest levels, in an extreme case the leaves of AVL tree can span the range beetwen levels *h* and 2*h*. The worst topology of the AVL tree with maximum height for a given number of nodes is called Fibonacci tree.

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

In spite of the fact that the AVL tree is only "nearly" balanced, the time complexity of the operations is $O(log\ n)$, where *n* is the number of nodes in the tree. It is demonstrated in [3] that for an AVL tree with *n* nodes its height *h* satisfies the condition

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

Consequently, the search, insertion, and delete operation, as well as finding the minimum, maximum, predecessor and successor, in the AVL tree have the *O(log n)* complexity, because their complexity is *O(h)* in a tree whose height is *h*, and each AVL with *n* internal nodes is a binary search tree of *O(log n)* height.

3. THE RED BLACK TREES

The red-black tree, just like the AVL tree, is a type of nearly balanced tree in which each node and branch are characterized by its colour. The coloring rules are stricly defined.

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

1. Every node is either red or black.
2. Every leaf (NIL) is black.
3. If a node is red, then its both sons are black.
4. Each path from the root to a leaf contains the same number of black nodes. [4]

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

The RB trees are used as an implementation of the 2-3-4 trees in a form of binary tree [8]. A significant characteristic of 2-3-4 trees is that all the leaves are at the same level, which ensures the optimal balance. Besides the usual 2-nodes with one key and two subtrees as in the binary trees, these trees may also have 3-nodes with two keys and three subtrees, as well as 4-nodes with three keys and four subtrees. The 2-3-4 trees are B-trees of degree 4 and are isomorphic with the RB tree [9], which means that for each 2-3-4 tree there is at least one RB tree as its binary representation [5]. This fact is important because the implementation of the 2-3-4 trees is rather complex, since in the operations, when necessary, one type of the node has to be transformed into another. These transformations are demanding and may affect the performance. In addition, the higher degree of the tree, the more memory is used. Therefore, its representation in a form of standard binary tree is quite attractive.

The idea of 2-3-4 tree implementation via RB tree consists in representing 3-node and 4-node by small binary trees connected with the red branches, while the black branches connect the original nodes of the 2-3-4 trees. The node that has a black branch with the parents is a black node, and the node that has the red branch with the parent is a red node.

The mapping of 2, 3 and 4- nodes of the 2-3-4 trees is performed as follows:

- 2-node is represented as a binary node with two black sons.
- 3-node is represented as two binary nodes, one red, and the other black, whereby the branch between the nodes is red. 3-node may be represented with two orientations which depend on the balancing requirements.
- 4-node is represented as the set of three binary nodes with the black node as a parent, while the sons are red. Two red branches are used.

The consequence of such node mapping, as well as the fact that all leaves are at the same height, ensures the four conditions of the RB definition are satisfied. It also ensures the relaxed balance of the tree, which in turn decreases the overhead of the tree maintenance.

If the tree structure is modified by some insert or delete operation which impairs the requirements of the definition, the tree transformations in form of rotations are performed in order to reestablish the correct structure and coloring. The basic operations of the RB trees have been described in [5], [6] and [7]. Just like the AVL trees, the logarithmic performance of these trees is also guaranteed, as it is demonstrated in [4]. It was proven that the RB tree with *n* internal nodes has a height no greater than $2 \cdot \log_2 (n+1)$. Therefore, since the usual operations in the RB tree with *n* nodes whose height is *h* are of order *O(h)*, it follows that its time complexity is *O(log n)*.

4. RELATED WORK

Several studies on the comparison of binary search trees with a relaxed balance criterion can be found in the open literature. Carlton examined the performance of the HB[k] trees, i.e. height balanced trees, where the AVL tree is a special case for k=1. The results of the simulations for evaluating the performance of the HB[k] tree were presented in [10]. It was concluded that the execution times of the procedures for maintaining the HB[k] trees, for k > 1, 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 trace back. The cost of maintaining HB[k] trees drops significantly as the allowed imbalance (k) increases. Both analytical and experimental results that show the cost of maintaining HB[k] trees as a function of k are discussed. As far as the AVL tree is concerned, 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. On average 0.465 rotation operations are required in case of insertion and 2.78 nodes are revisited in order to adjust the heights. Also, 0.214 rotations on average are necessary per delete operation and the average number of the revisited nodes when maintaining the height attributes is 1.91.

Bear and Schwab in [11] empirically compare the height of the balanced trees with the weight balanced trees by means simulation with a synthetic workload. In the conclusion, they give preference to the AVL trees.

Wright in [12] gives a comparative analysis of nine different types of trees, e.g. the AVL trees, several types of the weight balanced trees (trees of a limited balance, with coefficient $\alpha = 1 - \sqrt{2}/2$, described in [13]), the trees where the searched node moves by one level towards the root [14], as well as the appropriate combinations of some algorithms. The evaluation measures the execution time for different types of input sequences. The operations considered were insertion and searching, while deletion was not taken into account. It was concluded that the most efficient tree is a kind of the AVL tree augmented by certain operations.

The study presented in [15] is perhaps the closest to the concept of this paper, whereby some types of the AVL trees and several types of self-adjusting binary search trees ([16]) were considered and compared [16]. It was concluded that the AVL trees are the most efficient ones, and among the self adjusting structures, the splay trees give the best results [17] when top-down splaying operation [18].

In [19] three types of the binary search trees were also compared: the AVL, RB and splay trees. The performances of these trees when applied in the system software are compared, and therefore the tests were performed for various platforms. As several insertion operations were considered in the tests, the RB trees were more efficient for random data set, the AVL trees were better in the sorted data set. When it comes to search operation, the AVL trees are also more efficient.

The AVL and RB trees are also compared in [20]. Apart from these types of trees, skip lists and hash tables are also evaluated. The comparison is based on the efficiency of the stated structures in the dictionary implementation. The author gives the preference to the RB trees.

Although the studies from [15] and [19] are quite close to the topic of this paper, regardless of the same choice of the analyzed trees, the comparisons are made from different perspectives. While some previous studies (e.g., [19]) observe a specific environment in which the trees are applied, our study presents more general performance study independent on the implementation of algorithm, operative system, concrete application and the machine on which tests are conducted.

5. EVALUATION METHOD

The comparative performance evaluation of the AVL and RB trees has been made by the simulation method with specific synthetic workload. The characteristics of the workload are quite diverse concerning the parameters such as: the number of keys, the range of the key values, the distribution of the key values, time locality, relative frequency of search, insert and delete operations, probability of successful and unsuccessful search, etc. The performance indicators have been chosen to be independent on the algorithm implementation and platform on which the measurements are performed.

The simulation environment, as well as the algorithms of the tree operation itself, is implemented in *C* programming language, Microsoft Visual Studio 6.0. All experiments have been carried out on the WindowsXP platform, version 2002, on Pentium 4, 3.00GHz (Intel Pentium 4) with 1GB RAM.

The key sequences were generated in order to obtain more complete insight into the AVL and RB tree performance. Intervals from which the key values are taken, the number of elements in the sequence and the frequency of the key values were varied, whereby the care was taken to simulate the time locality of the keys.

The lengths of the key sequences are chosen to be between 10 and 1000000 elements in multiples of 10. The number of elements in the sequences was varied in order to establish how the performance depends on the number of keys.

The values in the key sequences may be repeated. Intervals from which the key values are taken and the size of the key sequence is varied, whereby the number of elements of the formed sequence is one of six stated values. Lower limits of the interval start from zero, and the upper limit varies from case to case. Two groups of the key sequences have been used here, and they differ according to the way of key generation.

In the first group, the keys are generated by *rand* method, whereby the elements are chosen out of a certain interval that changes, and the number of the sequence elements varies as well. The file names used to store these sequences indicate the random key distribution (the names start with R the interval the lower and upper limit, and the number of keys in the sequence. For example, R1000_100000.txt is a file which contains a sequence of 100000 elements, whose randomly selected key values are from the interval [0, 999]. The interval upper limits of the key value in the sequence belong to the set {10, 100, 1000, 10000, 30000, 40000, 100000, 1000000}. The consequence of such a key choice is that some values from the interval can be repeated, while some values do not appear in the sequences.

In the second group, the key values also can be are repeated in a sequence, the number of sequence elements is also varied, as well as the interval the key values, whereby the number and interval have the same values as in first group. The main difference is in the method of generaton of the next key in sequence. The goal of forming this sub group of key sequences was to obtain the key sequences with a non-uniform distribution and to express the temporal locality of chosen values which is sometimes characteristical for tree accesses. The function

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

was used in the following way to enforce different leves of temporal locality. An auxiliary initial sequence of keys without the repetition of values is formed (let it hereinafter be sequence *key*). Then, x is randomly generated from the interval $x \in [0,1]$. With such x, using function (1) y is calculated and it can be noticed that $y \in [0,1]$ for a>0. If the size of the *key* sequence is denoted by n, index i is now calculated as $i = n \cdot y$. Finally, element of the *key* sequence with index i is entered into the newly generated sequence. This PROCEDURE is repeated until the resulting sequence of the required length is generated.

By varying the parameter a, the curve's concavity can be adjusted, as shown in Figure 1. As the values of the parameter a are higher, the levels of the time locality of keys in the resulting sequence grows. Values 1, 10, 50 and 100 were taken for parameter a, and the results for a = 100 were analyzed. The size of the sequences, as well as the interval upper limit belongs to set {10, 100, 1000, 10000, 100000, 1000000}. Resulting sequences are stored into the files whose names are formed in the following way:

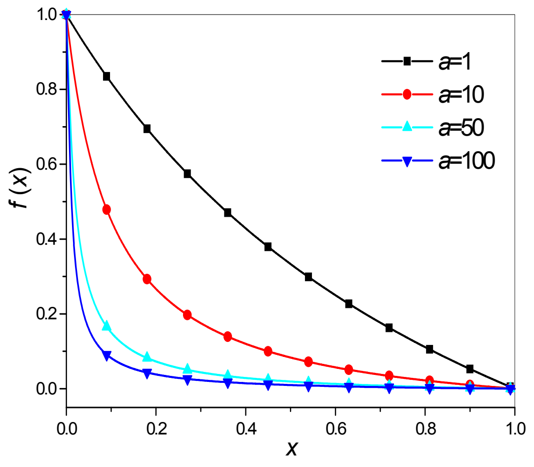

Figure 1: Function (1) values on the interval $x \in [0,1]$ for different values of parameter *a*.

nrIntervalUpperLimit__SequenceElementsNumber, e.g.

nr10000_100000.txt.

Such a synthetic workload model allows to evaluate how the number of the keys influences the efficiency of the series of operations on chosen tree types, and how the key repetition and the key values distribution influence the performance concerning each tree type.

The measurements of a series of three types of operations were considered: search, insertion and deletion of the key values. The percentage of search, insertion and deletion operations in the operation series was varied, and their relative ordering was randomly determined. Percentage was chosen so that the search is the most frequently, and the frequency of insertion and is the same. This choice is based on the fact that search is the most widespread operation in real conditions.

The series of tree operations were performed on the different initial trees. This paper presents the results obtained with initial tree generated from the values in the interval [0 .. 99999] inserted in random order. The cost of forming an initial tree is not considered in discussion of the results of a series of three types of operations, but their cost and choice will be discussed in the beginning of next section. The following performance indicators were collected:

- Average height during search operation – average height on which the element was found during successful search, or, the height of tree node in which the search was finished in the case of unsuccessful search.
- Tree height – this parameter refers to the maximum height of initial tree on which search series was performed.
- Average number of rotations per operation.

6. RESULTS AND DISCUSSION

In order to set up the scene, the initial tree should be built on which a some series of operation will be performed. Two ways of building an initial tree are examined: by insertion of sorted key values and by insertion of randomly chosen values. Figure 2 shows the height of resulting AVL and RB trees of ten up to a million nodes generated using the same keys inserted as a sorted sequence (I10 to IMillion) and as a random sequence (R10 to RMillion). It can be concluded from Figure 2 that both trees have lower heights for sorted order of inserted keys. In this case, the AVL trees have lower height than RB trees, and the difference between their heights is larger with the number of values inserted. If the keys are inserted in random order, the difference is smaller. In most cases even the heights are equal for both trees. The use of building the initial tree by inserting the keys from sorted sequence would bias the results of subsequent performance evaluation on series of operation where search operation prevails. In that case the AVL trees would have better performance; since the search tree operation does not require any rotation and it efficiency depends mainly on the tree height. Also, randomly generated initial tree is considered as much more typical case than the tree built from sorted sequence.

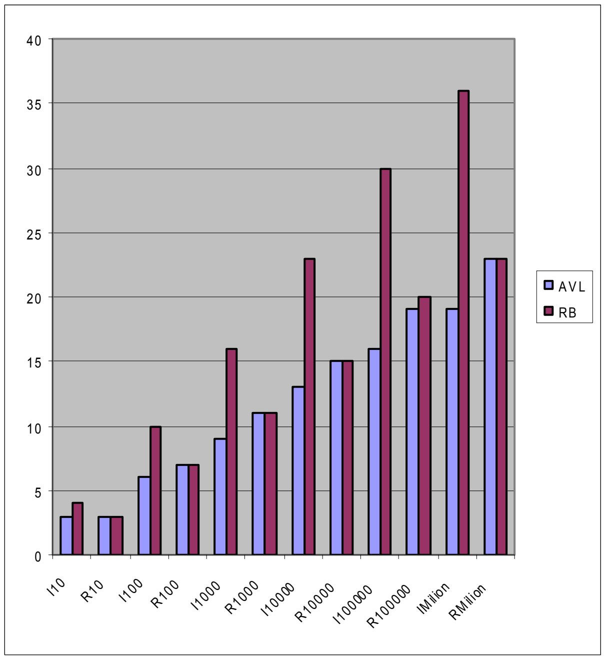

Figure 2: the height of the AVL and RB trees built with insertions of sorted and random ordered keys.

From the results given in Table 1, it is clear that, as expected, it takes more rotations to make the initial AVL tree then the RB tree. So building AVL tree is more expensive, but justified in case of sorted key insertions. This advantage increases with the number of search operation in the sequence of operations.

Table 1: Number of rotations in series of different number of insertion operations in random and sorted order of key values

Number of values	Number of rotations (insertions of random values)		Number of rotations (insertions of sorted values)	
	AVL	RB	AVL	RB
10	2	2	6	5
100	68	68	93	89
1000	646	646	990	983
10000	6452	6452	9986	9976
100000	64372	64372	99983	99969
1000000	640287	640287	999980	999963

There follows the results of the measurement on the series of operations with 10% insertions, 80% searches, 10% key deletions. The initial tree was built from a sequence of keys with 100,000 different values in random order, without repetition. The initial tree was taken because it is realistic that there is already some data in the memory, and therefore the deletion and search operations are not unsuccessful at the beginning of the series in a high percentage. The cost of building the initial tree has not been taken into account. Maximum height of the initial AVL tree was 19, and of the RB tree was 20. Maximum height of tree did not change during the performance of operations compared to the initial height regarding both types of trees. The number of operations is fixed at 100,000.

Table 2: Results of measuring the 100,000 operations series (10% insertions, 80% searches and 10% deletions) in key sequence whose values are randomly deployed in the sequence

File Name	Average height per operation		Average number of rotations per operation	
	AVL	RB	AVL	RB
R10_100000.txt	14,459	15,729	0,052	0,040
R100_100000.txt	14,231	15,594	0,056	0,046
R1000_100000.txt	12,716	13,618	0,058	0,048
R10000_100000.txt	14,174	15,168	0,066	0,059
R100000_100000.txt	14,795	14,864	0,083	0,073
RMilion_100000.txt	17,101	17,660	0,165	0,136

Table 2 data shows data for average height which is obtained by a series of operations and the average numbers of rotations per operation. A series of basic operations is performed on key sequences with random distribution where the keys may be repeated. The AVL trees appear to be more efficient. The RB trees have a smaller number of rotations per operation, but the difference is smaller than the difference in height, so that, although rotation is a more expensive operation than a level search (an additional recursive call or another repetition of iterative examination procedure), the efficiency is better for the AVL trees. It can be noticed that for both types of trees the increase of searched keys range, when it comes to height, does not lead to increase or decrease of average height at which it was arrived when performing operations. The reason is that during search, which is the most frequent operation and which represents the first phase of deletion and insertion operations, the tree topology is not modified, so the height at which it is arrived depends on the key position in the initial tree. The largest difference in the average height in both cases is in the case of RMilion_100000.txt input sequence. This is expected, considering that unsuccessful search appears here for keys greater than tree maximum values, and in this case the search ends in the leaf of the tree. As far as the rotations are concerned, it can be noticed that the average number of rotations per operation increases for both trees with an increase of range from which keys are searched.

Table 3: Series of 100,000 operations (10% insertion, 80% search and 10% deletion) on key sequence with non uniform distribution of key values

File Name	Average height per operation		Average number of rotations per operation	
	AVL	RB	AVL	RB
nr10_100000.txt	15,057	15,127	0,023	0,019
nr100_100000.txt	15,025	15,176	0,018	0,010
nr1000_100000.txt	14,841	15,899	0,009	0,006
nr10000_100000.txt	15,395	16,447	0,014	0,013
nr100000_100000.txt	15,265	15,292	0,023	0,021
nrMilion_100000.txt	16,961	17,168	0,060	0,050

Table 3 shows the results of the experiments for the same number and deployment of operations as in Table 1, but with non uniform key distribution which enforces the temporal locality. In this case, the AVL trees are more efficient than the RB trees, again. Same conclusions can be drawn as in the previous case when it comes to these two parameters in the function of value scope increase. The comparison of the results of Table 2 and Table 3 shows that for both trees the number of rotations decreases by more than 10% in case of non uniform sequences, but the heights are greater in this case. This is due to the fact that in second case the number of repetitions of the same key is greater than in files with random distributions. Therefore, the deletion and insertion operations are not performed because the value is already in the tree in case of the insertion or the value has already been removed in case of deletion, which reduces these operations to unsuccessful search avoiding the rotations in these cases.

Table 4: Results of measuring the series of 100,000 operations (25% insertion, 50% search and 25% deletion) on key sequence whose values are deployed randomly in the sequence

File Name	Average height per operation		Average number of rotations per operation	
	AVL	RB	AVL	RB
R10_100000.txt	14,189	15,897	0,021	0,015
R100_100000.txt	13,325	15,813	0,023	0,018
R1000_100000.txt	11,623	13,209	0,023	0,020
R10000_100000.txt	14,356	15,359	0,029	0,028
R100000_100000.txt	14,711	14,782	0,033	0,030
RMilion_100000.txt	17,925	18,017	0,066	0,055

Table 5: Results of the series of 100,000 operations (25% insertion, 50% search and 25% deletion) on key sequence with non uniform probability of key distribution

File Name	Average height per operation		Average number of rotations per operation	
	AVL	RB	AVL	RB
nr10_100000.txt	15,139	15,289	0,058	0,047
nr100_100000.txt	14,682	15,046	0,042	0,035
nr1000_100000.txt	14,686	15,794	0,018	0,014
nr10000_100000.txt	15,323	16,432	0,031	0,028
nr100000_100000.txt	15,382	15,409	0,042	0,037
nrMilion_100000.txt	16,416	16,836	0,130	0,109

Tables 4 and 5 show results of experiments for the same number of operations and an initial tree built in the same way as in previous experiments. However, the frequencies of the operations in the series are different. There are 25%, insertions, 50% searches, and 25% deletions. As in the previous case, the heights of trees after a series of operations for both the RB and AVL trees have not changed compared to the initial tree.

7. CONCLUSION

It goes for both types of trees that the sequence, number of repetitions and scope of keys do not significantly influence the performance of these trees.

The AVL trees are more efficient than the RB trees regardless of the key distribution and the number of their repetition. According to with the definitions of both trees, it is expected that the heights are lower in the AVL trees, i.e. that the number of comparisons is smaller in the case of AVL trees, and the number of rotations higher in order to keep this structure. Having in mind the cost (i.e. the complexity) of the rotation and the comparison (i.e. transition to the next level), and difference in the number of rotations, it can be concluded that in the case of AVL trees, these transformations lead to desired results. Therefore, that there are justified from the point of view of price of an average operation in the measured series of operations. It goes in favour of these trees that the realization of operations is simpler than in the RB trees.

The RB trees, although less efficient than AVL trees when search operations prevail, are not too far behind in the sense of measured parameters. They are significant since RB trees are used as a binary representation of 2-3-4 trees. As it was shown in paper [20], in some applications they give even better results than AVL trees.

In the cases when the distribution of keys is not uniform, and when the time locality of keys is increased, as was the case in many applications, neither tree type can provide an improvement compared to the case when the values appear randomly. This is a lack of both types of the considered trees since in both cases the operation efficiency to a great extent depends on the key deployment in the initial tree. The prospective future work on evaluation of these trees with the splay trees is appealing.

8. REFERENCES

[1] Tomašević M. 2011. Algorithms and Data Structures, *Academic mind*, Belgrade (in Serbian).

[2] Adelson–Velski G. M. and Landis E. M. 1962. An Algorithm for the Organization of Informations“, *Soviet Mathematics Doklady. 3: 1259-1263.*

[3] Živković M. 2000., Algorithms, *Matematički fakultet, Beograd.* (in Serbian).

[4] Cormen T., Leiseron Ch., Rivest R. 1990. Introduction to Algorithms, *The MIT Press, McGrawHill.*

[5] Flaming B. 1993. Practical Data Structures in C++, *John Wiley and Sons.*

[6] Knuth D. 1997., The Art of Computer Programing, *Vol. 3, Third Edition, Addison – Wesley.*

[7] Urošević D. 1996. Algorithms in C programming language, *Mikro knjiga.* (in Serbian).

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

[9] Bayer R. 1971. Binary B-Trees for Virtual Memory, Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, November 11-12.

[10] Karlton P.L., Fuller S.H., Scrogas R.E. and Kaehler E.B. 1976. Performance of heigh-balanced trees, *Communications of the ACM, vol 19, no1, pp23-28.*

[11] Bear J. L. and Schwab B. 1977. A Comparation of Tree-balancing Algorithms, *Comunications of the AC, 20(5):pp. 322-330.*

[12] Writght W. E. 1980. *An Empirical Evaluation of Algorithms for Dynamically Maintaining Binary Search Trees*, Proceedings of the ACM 1980 annual conference, pp. 505-515.

[13] Nievergelt J. and Reingold E.M. 1973. Binary Search Trees of Bounded Balance, *SIAM Journal Computing 2, pp 33-43.*

[14] Bell J. and Gupta G. 1993. An Evaluation of Self-Adjusting Binary Search Tree Technics, *Software -Practice and Experience, v23. pp. 369-8.*

[15] Bitner J. R. 1979. Heuristics That Dynamically Organize Data Structures, *SIAM J. Computing, 8, pp 82-110.*

[16] Allen B. and Munro I. 1978., Self-organising Binary Search Trees, JACM, 25, pp 526-535.

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

[18] Štrbac-Savić S., Tomašević M. 2012. Analiza tehnika za reorganizaciju samopodešavajućih stabala, *In Proceedings of the Infoteh.* 2012. (March 21-23. 2012.) Jahorina

[19] Pfaff B. 2004. Performance Analysis of BSTs in System Software, *ACM SIGMETRICS, pp410-411.*

[20] Neyer M. P. 2009A Comparison of Dictionary Implementations.
http://www.cs.unc.edu/~plaisted/comp750/Neyer%20paper.pdf