UDC 004.651:519.6 **doi: 10.26906/SUNZ.2026.1.131**

Anatolii Shostak

National Aerospace University “Kharkiv Aviation Institute”, Kharkiv, Ukraine

COMPARATIVE EXPERIMENTAL ANALYSIS OF METHODS FOR IMPLEMENTING THE AVL TREE

Abstract. The paper presents an experimental proof of the difference between two computational models of AVL trees: the classical online balanced AVL tree and the canonical median tree constructed on the basis of a sorted array of keys. Despite close asymptotic estimates of the complexity of search, insertion, and deletion operations, these implementations of AVL trees demonstrate fundamentally different behavior in practice. The study analyzes the impact of balancing strategy, tree shape, and search operation implementation method on the performance of search, key insertion, and deletion operations, as well as AVL tree creation. It is shown that using binary search on a sorted array of keys in a canonical AVL tree significantly improves the search time, but leads to linear complexity of insertion and deletion operations. The results confirm that the considered implementations are optimal for different load classes and are not interchangeable. The main contribution of this work is the experimental proof that canonical AVL trees and online AVL trees are fundamentally different computational models rather than competing implementations.

Keywords: computational model, AVL tree, binary search, asymptotic analysis.

Introduction

Height-balanced binary search trees are a fundamental data structure widely used in systems and applied programming. AVL trees [1–3], proposed by G. M. Adelson-Velsky and E. M. Landis, guarantee logarithmic tree height through a strict height-balanced condition and local rotations.

The classic implementation of an AVL tree [1-3] is focused on an online model in which each insertion or deletion operation immediately restores the tree's balance.

However, in practice, there are alternative scenarios for using AVL trees in which the tree is built in batches, and search operations dominate modification operations. Such scenarios arise in workloads with, for example, read-dominant data, static indexes, and autonomous analytics pipelines.

This paper considers two approaches:

• AVL(A) – a classical online AVL tree balanced by rotations;

• AVL(B) – a canonical AVL tree constructed using the median method from a sorted array of unique keys and completely rebuilt for each insertion and deletion.

AVL trees are traditionally analyzed in terms of the asymptotic complexity of operations, the number of rotations, and upper bounds on the height [1–3]. However, relatively few studies have been devoted to the experimental analysis of alternative implementations of AVL trees, in particular, canonical trees of a fixed shape constructed from sorted data, i.e., using data from preorder and inorder traversals of the tree.

Such approaches are more often considered in the context of static binary search trees (BSTs) or optimal search trees.

This work complements existing studies [4–6] by demonstrating practical differences between online and batch AVL tree balancing models.

The aim of the work is to experimentally compare the AVL(A) and AVL(B) approaches in terms of execution time of operations of creating an AVL tree, searching, inserting and deleting keys.

Main part

The classic AVL tree AVL(A) implements the standard AVL tree algorithm [1-3]: - insertion and deletion are performed recursively, - height balance is maintained through single and double rotations, - tree height is maintained within O(log n). This implementation is optimal for scenarios with frequent tree structure changes.

An AVL(A) tree depends on the insertion order of keys, allows multiple shapes for the same set of keys, and guarantees only height balance, not shape minimality. A canonical median AVL tree, AVL(B), is constructed as follows: - the input array of keys is cleared of duplicates, - the keys are sorted, - the tree is built recursively, choosing the median of the sorted array as the root.

The result is a canonical AVL tree of minimum height and minimum node depth variance among all binary search trees on the same set of keys.

A canonical tree for a set of keys means that it is a binary search tree that: - depends only on the set of keys, - does not depend on the order in which keys are inserted into the tree, - has the minimum possible height, and - can be reconstructed from the set of keys in a unique way.

A key feature of the implementation is the storage of a sorted array of keys, which is used not only to construct the tree but also to perform search operations via binary search. Insertion and deletion operations are implemented as complete tree rebuilding.

For the experiments, arrays of unique integer keys, generated uniformly at random within a given range, were used. Input data sizes ranged from 5,000 to 25,000 elements.

The following operations were evaluated: Create – constructing a tree from an array of keys, Insert – sequentially inserting elements, Delete – sequentially deleting elements, and Search – searching for keys.

For the Delete operation, two scenarios were considered: 1) a complete deletion of all tree keys, 2) a partial deletion of tree keys (after deleting nodes, the tree remained non-empty).

For the Search operation, three scenarios were considered: 1) all searched keys are present in the tree,

2) none of the searched keys are present in the tree, 3) half of the searched keys are present in the tree and the other half are not.

The following metrics were recorded in the experiments: - operation execution time, - tree height after the operation, - average tree node depth, - tree node depth variance, - number of rotations (for AVL(A)).

Experiments with the Create operation for random keys uniformly distributed in the range (0, 50000) showed (Table 1, Fig. 1–4) that AVL(B) significantly outperforms AVL(A) in tree construction time. This is explained by the linear complexity of median construction and the absence of rotations in AVL(B).

Table 1 – **Experiment results for the Create operation**

type	n	tMs	h	avgD	varD	leftR	rightR
AVL(A)	5000	2,51	13	10,48	2,69	1664	1663
AVL(B)	5000	2,08	12	10,28	2,17	0	0
AVL(A)	10000	5,27	14	11,42	2,76	3161	3164
AVL(B)	10000	4,2	13	11,19	2,14	0	0
AVL(A)	15000	8,19	15	11,95	2,88	4516	4514
AVL(B)	15000	6,00	13	11,74	2,18	0	0
AVL(A)	20000	11,22	15	12,31	2,83	5755	5756
AVL(B)	20000	7,64	14	12,01	2,00	0	0
AVL(A)	25000	14,29	16	12,57	2,82	6872	6868
AVL(B)	25000	9,66	14	12,34	2,21	0	0

In Table 1, type is the type of AVL tree, n is the size of the key array, tMs is the operation execution time in Ms, h is the height of the tree after the operation is executed, avgD is the average depth of tree keys, varD is the depth variance, leftR, rightR are the number of left and right rotations, respectively, for AVL(A).

AVL(B) minimizes tree height globally, while AVL(A) minimizes it only locally (Table 1, Fig. 1). Moreover, AVL(A) has a consistently higher tree height, and compared to AVL(B), the difference in heights increases with n.

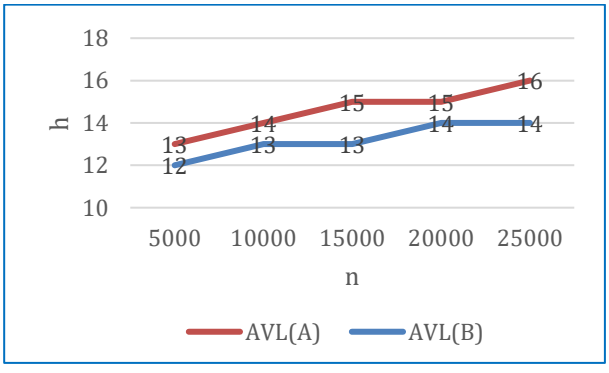


Fig. 1. Dependence of the tree height h on the number of elements n in the tree

The AVL(B) tree is constructed faster, while the AVL(A) tree must spend significant time on comparisons, height determinations, and rotations (Table 1, Fig. 2). For AVL(A), the number of left and right rotations is approximately equal and increases approximately linearly with the number of keys *n* (Table 1).

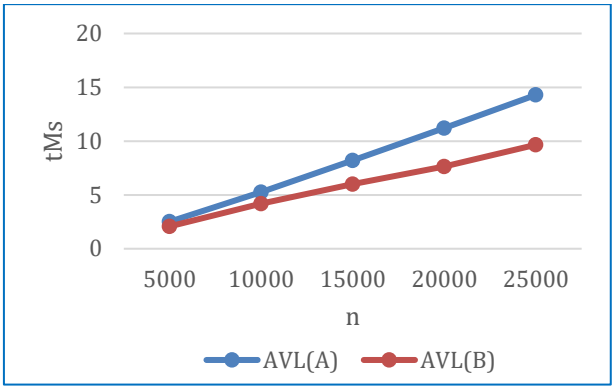


Fig. 2. Dependence of the execution time of the Create operation on n

The average key depth avgD for AVL(B) is smaller than for AVL(A) for all n (Table 1, Fig. 3). This will ensure that AVL(B) requires fewer comparisons during search.

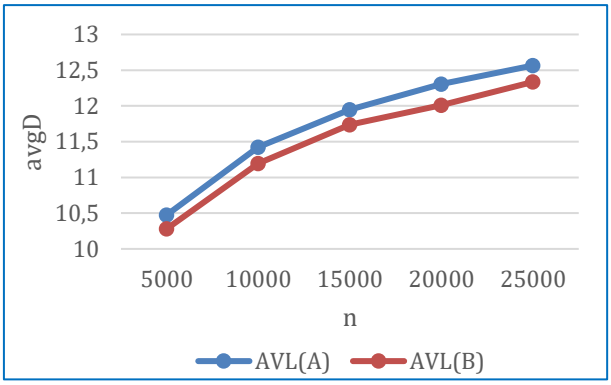
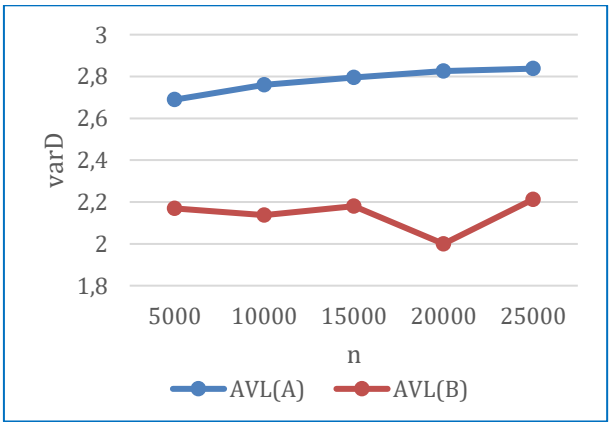


Fig. 3. Dependence of the average key depth avgD on n

The variance of key depth varD is lower and more stable for AVL(B), while it is higher for AVL(A) (Table 1, Fig. 4), since the shape of the AVL(A) tree depends on the insertion order of keys.



For insertion and deletion operations, AVL(A) demonstrates significantly better performance (Table 2). Despite the poorer tree structure and the presence of rotations, the O(log n) asymptotics of operations ensures fast execution time.

Table 2 – **Ratio of times (tMs AVL(B) / tMs AVL(A)) for executing an insert operation and two scenarios for the delete operation**

n	Ratio of times		
	Insert	Delete	
		1)	2)
5000	658	1122,4	1483,4
10000	1864,1	2027,8	2480,3
15000	2398,6	2567,9	3153,2
20000	3133,3	3151,6	3654,4
25000	3656,1	3696,4	4024,2

AVL(B), despite its more compact tree form (lower height and average depth), exhibits linear complexity of O(n) operations, since each operation results in a complete tree rebuild.

The dependence of the ratio of insertion and deletion times on n for the two approaches, AVL(B) and AVL(A), respectively, shows a linear increase with increasing n and a significant advantage for the AVL(A) approach (Table 2, Fig. 5 and 6).

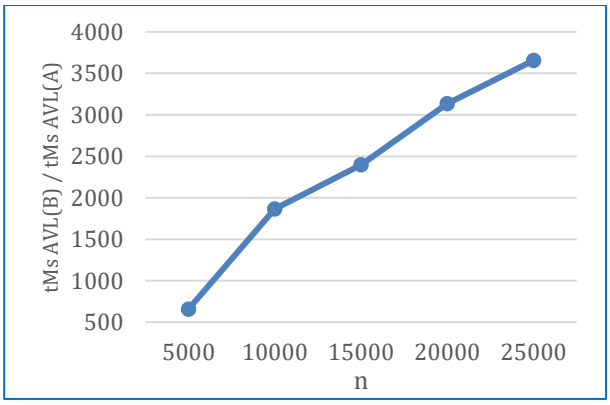


Fig. 5. Dependence of the ratio of execution times of the insert operation on n for two approaches AVL(B) and AVL(A)

For all dimensions n, scenario 1) (complete deletion of all tree keys) shows a lower time ratio for the two approaches, AVL(B) and AVL(A), respectively, compared to scenario 2) (partial deletion of tree keys) (Table 2, Fig. 6).

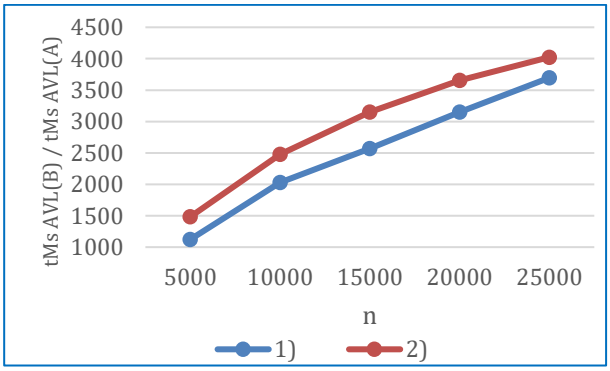


Fig. 6. Dependence of the ratio of times for two scenarios of performing the deletion operation on n for two approaches AVL(B) and AVL(A)

Although deletion in AVL(A) is algorithmically more complex than insertion, its cost remains logarithmic due to local rebalancing. In contrast, AVL(B) performs a complete rebuild for these operations; however, deletion additionally destroys the structural locality accumulated during construction, resulting in poor cache performance and higher constant coefficients.

Three scenarios were considered for the search operation:

1) all searched keys are present in the tree – search for k=1000 keys, where p=500 keys are present in the tree and m=500 are not;

2) none of the searched keys are present in the tree – k=1000, p=0, m=1000;

3) half of the searched keys are present in the tree and the other half are not – k=1000, p=500, m=500.

When using search, the AVL(A) and AVL(B) approaches demonstrate comparable times (Table 3, Fig. 7), with the differences between the search scenarios being virtually indistinguishable.

Using binary search over the underlying sorted key array in the AVL(B) search operation results in a noticeable speedup. This approach preserves the tree structure for structural analysis.

Table 3 – **Ratio of execution times (tMs AVL(B) / tMs AVL(A)) for three search operation scenarios**

n	Search scenario		
	1)	2)	3)
5000	1,05	1,04	1,02
10000	1,02	1,15	1,08
15000	0,97	0,96	0,92
20000	0,98	0,93	1,01
25000	1,03	1,09	0,98

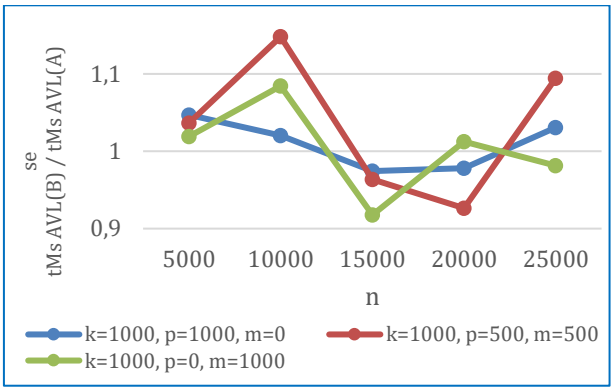


Fig. 7. Ratio of times for three scenarios of performing a search operation from n for AVL(B) and AVL(A)

AVL(B) has a strictly minimal height and the smallest depth variance (Table 1). AVL(A), despite meeting the balancing conditions, produces trees with greater shape variability, which can negatively impact access locality and search time (Tables 1, 3).

The results show (Tables 1-3) that the choice of AVL tree implementation should be determined by the nature of the workload:

• for dynamic scenarios with frequent insertions and deletions, the classic online implementation is preferred;

• for static or search-oriented scenarios, the canonical AVL tree with binary array search provides better performance and predictability.

Thus, AVL(A) and AVL(B) represent different algorithmic paradigms, not competing optimizations of the same data structure.

Conclusions and future directions

This paper presents a systematic experimental comparison of two conceptually different approaches to AVL tree construction and maintenance: the classical online balanced AVL tree (AVL(A)) and the canonical median-based AVL tree with complete rebuilding (AVL(B)).

Experimental results show that, despite identical asymptotic search complexity, the two approaches exhibit fundamentally different performance characteristics due to their underlying computational models. AVL(A) maintains balance gradually through local rotations and is therefore optimal for dynamic workloads dominated by insertion and deletion operations. In contrast, AVL(B) constructs a canonical tree using sorted keys and completely avoids rotations, achieving minimal variance in tree height and depth.

The key result of this work is that replacing the tree-based search in AVL(B) with binary search over an underlying sorted array of keys significantly improves search performance, making it comparable to or faster than AVL(A). This highlights the importance of separating the logical tree structure from the physical data representation used to execute queries.

At the same time, the batch rebuild strategy used by AVL(B) results in insertions and deletions in linear time, making it unsuitable for workloads with intensive online updates. However, in search-dominant or static scenarios, AVL(B) provides excellent structural optimality and predictable performance characteristics.

Overall, the results confirm that AVL(A) and AVL(B) are not competing implementations of the same data structure, but rather represent distinct algorithmic paradigms optimized for different workload profiles. Therefore, the choice between them should be determined by the relative frequency of update and query operations, as well as the requirements for determinism and structural canonicity. Future research directions include experimentally studying the influence of the AVL tree shape and its height on search time while maintaining balancing conditions, and considering hybrid approaches combining the canonical form and partial online balancing.

Use of Artificial Intelligence Tools. The author confirm that artificial intelligence technologies were not used in the creation of the presented work.

REFERENCES

1. Adelson-Velskii M., Landis E.M., An algorithm for the organization of information. Proceedings of the USSR Academy of Sciences, 1962. 146. – pp.263–266. URL: https://zhjwpku.com/assets/pdf/AED2-10-avl-paper.pdf
2. Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., Clifford Stein. Introduction to algorithms. – MIT Press, 2022. – 1312 pp. URL: https://www.cs.mcgill.ca/~akroit/math/compsci/Cormen%20Introduction%20to%20Algorithms.pdf
3. Chauhan S., Thakur S., Rana S., Sharma S. A brief study of balancing of AVL tree. International Journal of Research (IJR). Vol. 1, Issue 11, 2014, pp. 406-408, URL: https://scispace.com/pdf/a-brief-study-of-balancing-of-avl-tree-53a0i1etow.pdf
4. Sedgewick R., Wayne K., Algorithms: / Sedgewick R. Princeton University, Addison-Wesley, 2011. 955 pp. URL: https://algs4.cs.princeton.edu/home/
5. Wiener R. AVL Trees //Generic Data Structures and Algorithms in Go: An Applied Approach Using Concurrency, Genericity and Heuristics. Berkeley, CA : Apress, 2022. C. 315-347, https://doi.org/10.1007/978-1-4842-8191-8_10
6. Bounif L., Zegour D. E. Toward a unique representation for AVL and red-black trees. Computación y Sistemas, Vol. 23, No. 2, 2019, pp. 435–450, doi: https://doi.org/10.13053/CyS-23-2-2840

Received (Надійшла) 25.11.2025
Accepted for publication (Прийнята до друку) 11.02.2026
Publication date (Дата публікації) 27.02.2026

ВІДОМОСТІ ПРО АВТОРІВ / ABOUT THE AUTHORS

Шостак Анатолій Васильович – кандидат технічних наук, доцент, доцент кафедри комп'ютерних систем, мереж і кібербезпеки Національного аерокосмічного університету України «Харківський авіаційний інститут», Харків, Україна;
Anatolii Shostak – Candidate of Technical Sciences, Associate Professor, Associate Professor of a Department of Computer Systems, Networks and Cybersecurity of the National Aerospace University of Ukraine "Kharkiv Aviation Institute", Kharkiv, Ukraine;
e-mail: a.shostak@csn.khai.edu; ORCID Author ID: https://orcid.org/0009-0003-7687-3641

Порівняльний експериментальний аналіз способів реалізації АВЛ-дерева

А. В. Шостак

Анотація. У роботі проводиться експериментальний доказ відмінності двох обчислювальних моделей AVL-дерев: класичного онлайн-балансованого AVL-дерева та канонічного медіанного, що будується на основі відсортованого масиву ключів. Незважаючи на близькі асимптотичні оцінки складності операцій пошуку, вставки та видалення, дані реалізації AVL-дерев демонструють принципово різну поведінку на практиці. В рамках дослідження аналізується вплив стратегії балансування, форми дерева та способу реалізації операції пошуку на продуктивність операцій пошуку, вставки та видалення ключів, а також створення AVL-дерева. Показано, що використання бінарного пошуку за відсортованим масивом ключів у канонічному AVL-дереві суттєво покращує час пошуку, проте призводить до лінійної складності операцій вставки та видалення. Результати підтверджують, що аналізовані реалізації оптимальні для різних класів навантажень і не взаємозамінні. Основний внесок цієї роботи полягає в експериментальному доказі того, що канонічні AVL-дерева та онлайн-AVL-дерева є принципово різними обчислювальними моделями, а не конкуруючими реалізації.

Ключові слова: обчислювальна модель, AVL-дерево, бінарний пошук, асимптотична оцінка.