JURNAL TEKNOLOGI DAN OPEN SOURCE
Vol. 9, No. 1, June 2026, pp. 64~78
e-ISSN: 2622-1659, accredited Four Grade by Kemenristekdikti, Decree No: 152/E/KPT/2023
DOI: https://doi.org/10.36378/jtos.v9i1.5628

Empirical Performance Analysis of BST and AVL Tree on Modern Computing Architectures: A Stress Test Study Under Varying Data Distributions

Hazna At Thooriqoh[1], Ibnu Khoirul Anwar[2], Dimas Nugroho Dwi Seputro[3]
[1,2]Faculty of Computer Science, Universitas Pembangunan Nasional "Veteran" Jawa Timur, Indonesia
[3]Faculty of Economic and Business, Universitas Pembangunan Nasional "Veteran" Jawa Timur, Indonesia

Article Info

Article history:

Received 05 02, 2026
Revised 05 20, 2026
Accepted 06 06, 2026

Keywords:

AVL Tree
Binary Search Tree
Data Distribution
Game Leaderboard
Performance Analysis
Stress Test

ABSTRACT

Binary Search Tree (BST) and AVL Tree are fundamental data structures widely used for dynamic data management in performance-critical systems. Although both structures offer efficient theoretical complexity, their practical performance on modern systems is highly influenced by data distribution and workload characteristics. This study presents an empirical performance evaluation of BST and AVL Tree using a stress-test approach based on a game leaderboard system as a representative case study. Multiple workload patterns were simulated, including random, sequential (ascending and descending), and clustered data distributions, to reflect realistic high-frequency updates commonly observed in modern applications. Experimental results show that BST achieves slightly better performance under random data distributions due to the absence of balancing overhead. However, BST experiences severe performance degradation under sequential inputs, where it degenerates into an unbalanced structure. In contrast, the AVL Tree consistently maintains logarithmic height, achieving speedups of up to 32x compared to BST in worst-case scenarios. These findings indicate that while BST can be effective under controlled average-case conditions, AVL Tree provides superior robustness and predictable performance under non-uniform and adversarial workloads. For modern high-load systems such as game leaderboards, the balancing overhead of AVL Tree represents a minimal trade-off compared to the substantial stability and performance guarantees it offers.

Corresponding Author:

Hazna At Thooriqoh
Department of Computer Science
Universitas Pembangunan Nasional "Veteran" Jawa Timur
Surabaya, Indonesia
Email: hazna.thooriqoh.fasilkom@upnjatim.ac.id

1. Introduction

Efficient data organization and retrieval are fundamental requirements in modern information systems, particularly in applications that demand real-time data processing such as online gaming platforms, recommendation systems, financial transaction systems, and large-scale information retrieval services. As data volume and access frequency continue to grow, the choice of underlying data structures becomes a critical factor influencing system performance, scalability, and user experience [1].

Tree-based data structures have long been used to manage ordered data efficiently. Among these, Binary Search Tree (BST) is one of the most fundamental structures due to its simple implementation and intuitive design. Under ideal conditions, BST supports search, insertion, and deletion operations with an average time complexity of O(log n). However, BST does not impose structural constraints to maintain balance, making it highly sensitive to input order. When data are inserted in sorted or nearly sorted order, BST may degenerate into a linear structure with O(n) time complexity, leading to severe performance degradation. This behavior has also been reaffirmed in recent academic studies, which show that BST performance deteriorates drastically under ordered data compared to self-balancing trees such as AVL, particularly in modern computing environments [2]

To address this limitation, self-balancing tree variants have been proposed. The AVL Tree, introduced by Adelson-Velsky and Landis, enforces strict height balance through rotation operations after insertion and deletion [3]. This mechanism guarantees a worst-case time complexity of O(log n) for all fundamental operations, at the cost of additional computational overhead during updates. Previous studies have shown that AVL Trees often outperform unbalanced BSTs in adversarial or unpredictable workloads, while potentially incurring higher costs under purely random data distributions [4].

Despite extensive theoretical analysis of BST and AVL Tree, real-world system behavior often deviates from theoretical assumptions. Modern applications rarely operate under perfectly random input conditions. Instead, they exhibit dynamic and skewed data patterns, such as sequential score updates, clustered insertions, or bursty workloads. For example, in game leaderboard systems, player scores are frequently updated in monotonic or near-monotonic patterns, which can expose worst-case behaviors in unbalanced trees [5]. Consequently, empirical evaluation under realistic and stress-test scenarios is essential to complement theoretical complexity analysis.

Several prior studies have evaluated BST and balanced trees in isolation or within specific domains such as caching systems, CPU scheduling, and distributed query processing [6]. However, many of these studies focus either on theoretical complexity or on synthetic benchmarks without direct linkage to application-level systems. There remains a gap in empirical studies that systematically analyze BST and AVL Tree performance using real implementation data derived from complete system prototypes.

Motivated by this gap, this research presents an empirical performance analysis of BST and AVL Tree implementations using a game leaderboard system as a representative application case. The study employs stress testing under varying data distributions, including sequential, random, and mixed workloads, to evaluate execution time, structural growth, and operational stability. By grounding the analysis in data obtained from an implemented system rather than purely synthetic models, this work aims to provide practical insights for system designers and software engineers when selecting appropriate data structures for performance-critical applications. The main contributions of this study are threefold: (1) an empirical evaluation of BST and AVL Tree based on a fully implemented leaderboard system, (2) a systematic stress-test analysis under multiple data distributions, and (3) practical design recommendations for performance-critical applications.

This paper contributes a unified analytical and empirical framework for evaluating BST and AVL Tree performance under realistic workloads. The proposed model characterizes execution time as a function of height growth and workload distribution, enabling a principled explanation of observed speedups in adversarial scenarios. To the best of our knowledge, this is one of the first studies to explicitly connect analytical cost formulations with empirical leaderboard system behavior.

Recent studies have also explored the role of tree-based data structures beyond pure algorithmic efficiency, particularly in educational, system-level, and hardware-aware contexts. Rojas-Salazar et al. investigated the use of serious games to teach Binary Search Tree and AVL Tree concepts, demonstrating that game-based visualization can significantly improve understanding of tree rotations, balance properties, and structural evolution during insertions and deletions [7]. Although their focus is pedagogical, this work highlights the strong conceptual alignment between tree operations and game mechanics, reinforcing the relevance of tree-based structures in interactive systems such as game leaderboards.

From a system perspective, Kushwah et al. conducted an empirical study on proxy caching systems using LRU combined with BST and AVL Tree structures [6]. Their results show that AVL-based caching mechanisms provide more stable access time under dynamic and non-uniform access patterns compared to BST-based approaches. This finding is particularly relevant to game-like workloads, where access patterns are highly dynamic and skewed, supporting the argument that AVL Trees offer superior robustness in real-world systems with fluctuating workloads.

More recent research has shifted attention toward hardware-aware and architecture-level optimizations of tree-based data structures. Biebert et al. proposed a set of general techniques for optimizing tree-based data structures on heterogeneous memory systems by reordering tree nodes in memory without changing the logical structure of the tree [8]. Their experimental results demonstrate performance

improvements of up to 95% through cache-aware and memory-layout optimizations. This work is relevant to the present study as it emphasizes that traversal depth and access locality—both directly influenced by tree height—have a substantial impact on real execution time on modern hardware.

Concurrency and scalability of tree-based structures have also been extensively studied. Wang et al. proposed a concurrent update strategy for persistent randomized Binary Search Trees, enabling efficient multithreaded updates while preserving consistency and scalability on multicore systems [9]. While their work focuses on randomized and persistent BST variants rather than AVL Trees, it demonstrates that modern system demands increasingly require tree structures to be evaluated not only for single-threaded performance but also for concurrent execution characteristics.

At the microarchitectural level, Kim et al. analyzed how different memory access behaviors interact with CPU cache hierarchies and memory schedulers [10]. Their findings show that workloads with irregular and deep memory access patterns are more sensitive to cache misses and memory contention. This insight is directly applicable to tree traversal operations, where deeper and more irregular tree structures—such as degenerated BSTs—are more likely to incur cache inefficiencies compared to balanced trees.

Finally, Williams et al. introduced the Roofline performance model as a visual and analytical tool to reason about the upper bounds of application performance on multicore architectures [11]. Although not specific to tree-based data structures, the Roofline model provides a useful framework for interpreting whether observed performance bottlenecks stem from algorithmic inefficiencies or hardware limitations. This model can be leveraged to contextualize the empirical performance differences observed between BST and AVL Tree implementations on modern computing platforms.

2. Research Method

The methodological design of this study is further grounded in prior research that emphasizes the robustness of AVL-based indexing and the limitations of unbalanced BSTs under realistic system conditions. Khutwad and Pal demonstrated that AVL Tree–based indexing is highly effective for query processing in distributed environments, as it maintains logarithmic query performance even when data are partitioned across multiple nodes [12]. Their findings support the use of AVL Tree as a reliable baseline structure for performance evaluation in systems that resemble distributed or large-scale applications, reinforcing its suitability for modeling leaderboard-like data access patterns.

In contrast, Hirata and Nunome analyzed the performance of parallel and speculation-based construction of Binary Search Trees and identified significant bottlenecks when BSTs are built under skewed or non-uniform input distributions [13]. Their results indicate that even with parallelization, BST performance remains highly sensitive to insertion order, often leading to irregular tree structures and degraded efficiency. This insight motivates the inclusion of skewed and ordered data scenarios in the experimental methodology of this study to expose worst-case and stress-test behaviors of BST implementations.

Beyond algorithmic and system-level considerations, the methodological relevance of performance evaluation in leaderboard systems can also be justified from a user-centric perspective. Hohensinn et al. showed that ranking-based performance displays reduce users' cognitive processing of underlying information, enabling faster decision-making and perception when rankings are computed and presented efficiently [14]. This psychological perspective provides an additional rationale for emphasizing low-latency data structure operations in leaderboard systems, as faster search and update operations directly contribute to timely ranking updates that align with users' cognitive expectations and interaction patterns.

2.1. Case Study Design

A game leaderboard system is used as the experimental case study. Each player is represented by a unique score serving as the primary key in the tree, along with associated metadata such as player name. The leaderboard system supports core operations including insertion of new players, score updates, deletion, search by score, search by name, and ordered traversal for ranking display. This design reflects realistic use cases commonly found in online game systems.

Figure. 1 shows the operational flow of the implemented leaderboard application. The flowchart describes program initialization, the main menu loop, input validation, and the set of supported operations (e.g., switch mode BST/AVL, insert/delete, update score, searches, traversals, top-N view, import/export, and clear-all). The diagram is derived from the final project implementation and is used to clarify how user operations map to underlying tree procedures.

2.2. Implementation Details

The implementation is adapted directly from the final project used in this study. Both BST and AVL Tree are implemented in C++ using identical node structures and interfaces, differing only in the balancing logic. Each node stores the player score, player name, child pointers, and height information. For the AVL

Tree, balance factors are calculated after every insertion and deletion, and appropriate rotations (LL, RR, LR, RL) are applied when imbalance is detected.

The system includes additional analytical features such as automatic height calculation, node count, minimum and maximum score detection, and average score computation. These features are utilized to support deeper performance analysis during stress testing.

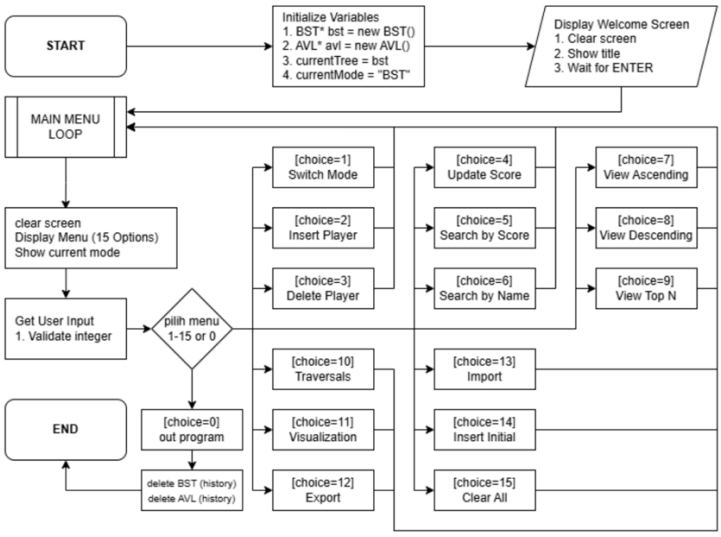

Figure. 1. Leaderboard application flowchart

2.3. Stress Test Scenarios

Stress testing is performed using scenarios derived from the final project report. Four data distributions are evaluated:

1) Random distribution, where player scores are generated uniformly at random.
2) Ascending distribution, where scores are inserted in strictly increasing order.
3) Descending distribution, where scores are inserted in strictly decreasing order.
4) Clustered distribution, where scores are concentrated within specific value ranges to simulate score farming or ranking congestion.

For each scenario, large-scale insert operations are executed, followed by search and traversal operations. Tree height and execution time are recorded after each test to observe structural growth and performance trends.

2.4. Evaluation Metrics

Performance evaluation focuses on execution time of insertion and search operations, tree height growth as the number of nodes increases, and overall operational stability. Tree height is used as a key indicator of structural efficiency, as it directly affects time complexity. These metrics are collected consistently across both BST and AVL Tree implementations to ensure fair comparison.

To provide a clear overview of the experimental pipeline, Figure. 2 presents the research workflow adopted in this study, from objective definition and prototype implementation to stress-test execution, metric collection, analysis, and reporting.

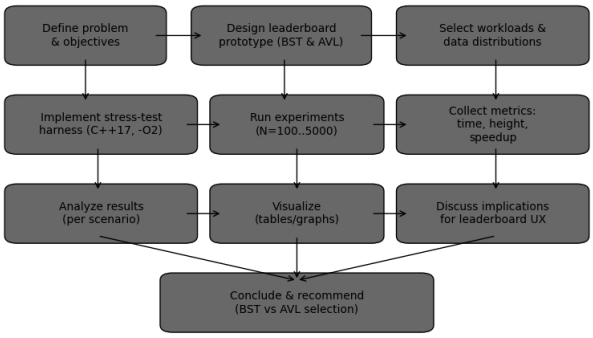

Figure. 2. Research workflow for empirical evaluation

3. Result and Discussion

This section presents and discusses the empirical results obtained from stress testing the BST and AVL Tree implementations. All experimental data are derived directly from the final project implementation and measurements, ensuring that the analysis reflects actual system behavior under realistic workloads.

3.1. Sequential Insertion Performance (Worst-Case Scenario)

The sequential insertion scenario represents the worst-case input pattern for a Binary Search Tree, where keys are inserted in strictly increasing order. Under this condition, BST fails to maintain structural balance and degenerates into a linear chain, effectively behaving as a linked list. This behavior is clearly reflected in both structural and performance metrics obtained from the experiment.

As shown in the experimental results, BST height grows almost linearly with the number of inserted nodes, reaching a height of 4,999 when the data size is 5,000. In contrast, the AVL Tree consistently maintains a logarithmic height, stabilizing at a height of 12 for the same data size. This observation confirms the theoretical height bounds of BST (O(N)) and AVL Tree (O(log N)) under adversarial input distributions.

The impact of this structural degradation is directly observable in execution time. While BST insertion time increases dramatically as the data size grows, AVL Tree exhibits a significantly slower growth rate. At N = 5,000, BST requires 1,112.56 ms to complete insertion, whereas AVL Tree completes the same operation in only 34.78 ms. This corresponds to a speedup of approximately 32× in favor of the AVL Tree.

The speedup trend increases sharply with data size, indicating that the performance gap between BST and AVL Tree expands superlinearly as N grows. This phenomenon occurs because each insertion into a skewed BST requires traversing nearly the entire tree, resulting in cumulative insertion costs approaching $O(N^2)$. In contrast, AVL Tree maintains bounded height through rotations, ensuring that each insertion remains O(log N).

These results demonstrate that sequential insertion patterns severely compromise the practical usability of the Binary Search Tree (BST) in real-time systems. When data is inserted in a strictly increasing or decreasing sequence, the BST gradually loses its balanced structure and begins to resemble a linear linked list. As a consequence, operations that are expected to be efficient, such as insertion, search, and update, experience a significant decline in performance. This degradation becomes increasingly evident as the dataset grows, causing execution times to rise dramatically. In real-time environments, where responsiveness and consistency are essential, such behavior can create serious performance bottlenecks and reduce the overall effectiveness of the system.

For applications such as game leaderboards, where score updates may occur in monotonic or near-monotonic order, the impact of this performance degradation becomes particularly problematic. The experimental results indicate that processing delays can exceed one second even at moderate data sizes. In practical terms, these delays would negatively affect user interactions, resulting in slower updates, reduced responsiveness, and an overall decline in user experience. Since modern users generally expect immediate feedback from interactive systems, delays of this magnitude are often considered unacceptable. By contrast, the AVL Tree maintains a balanced structure through automatic rebalancing operations, ensuring that its height remains logarithmic regardless of insertion order. This property enables the AVL Tree to deliver predictable and stable performance even under unfavorable input conditions, making it a more robust and scalable data structure for real-time applications.

Overall, this scenario empirically validates that structural balance is a critical factor in sustaining performance under adversarial workloads. The findings clearly show that maintaining balance directly contributes to operational efficiency and long-term scalability. Although the BST offers advantages in terms of implementation simplicity and relatively low overhead under favorable conditions, its vulnerability to worst-case insertion patterns significantly limits its reliability. Therefore, for production systems where data order cannot be strictly controlled or predicted, relying on a standard BST presents considerable risks, making self-balancing structures such as AVL Trees a more suitable and dependable choice.

Table 1. Comparison between BST and AVL Tree

Test	Data Size	BST Time (ms)	AVL Time (ms)	BST Hgt	AVL Hgt	Speedup (x)
Seq	100	0,45	0,52	99	6	0,87
Seq	500	11,23	2,89	499	8	3,88
Seq	1000	44,67	6,12	999	9	7,3
Seq	2500	278,34	16,45	2499	11	16,92
Seq	5000	1112,56	34,78	4999	12	31,99

Table. 1 illustrates the execution time comparison between BST and AVL Tree under sequential insertion. The figure clearly shows a steep linear increase in BST execution time as data size grows, whereas AVL Tree exhibits a much slower growth rate.

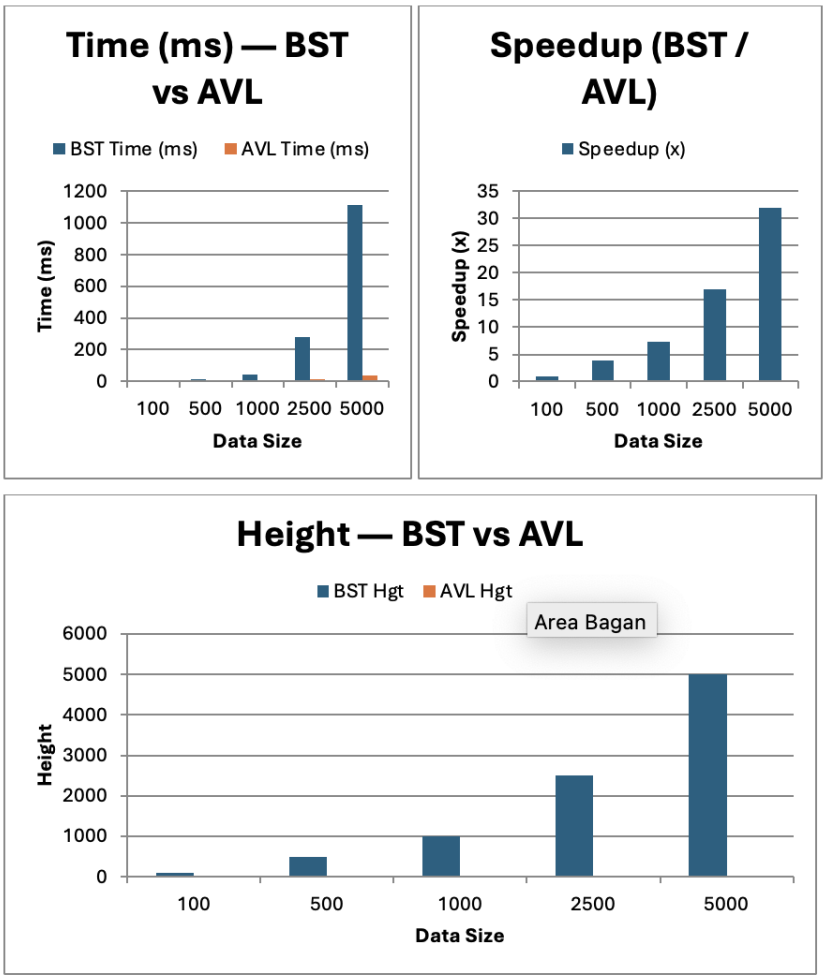

Figure 3. The Corresponding Tree Height Comparison

Fig. 3 presents the corresponding tree height comparison. BST height increases almost linearly with the number of nodes, reaching a height of 4,999 at N = 5,000, effectively degenerating into a linked list. In contrast, AVL Tree height remains bounded and grows logarithmically, stabilizing at a height of 12.

3.2. Random Insertion Performance (Average-Case Scenario)

The random insertion scenario represents the average-case workload for both Binary Search Tree (BST) and AVL Tree, where input keys are inserted in an unpredictable order. Under this condition, BST is less likely to degenerate into a skewed structure, allowing its inherent simplicity and lower operational overhead to be reflected in practical performance.

The experimental results indicate that BST consistently achieves lower insertion time than AVL Tree across all tested data sizes. At N = 100, BST completes insertion in 0.38 ms compared to 0.56 ms for AVL. This trend persists as the dataset grows, with BST requiring 31.89 ms at N = 5,000, while AVL requires 41.23 ms. On average, BST is approximately 23–25% faster than AVL under random insertion workloads. This performance advantage arises from the absence of balancing operations in BST, whereas AVL must perform height checks and rotations after each insertion.

Speedup measurements further confirm this observation. The speedup ratio (BST/AVL) remains consistently below 1.0, ranging from 0.68 to 0.77 across all tested data sizes. Unlike the sequential insertion scenario, where speedup increases sharply with N, the speedup in this scenario remains relatively stable. This indicates that the performance gap between BST and AVL does not widen as the dataset grows under random

input conditions, highlighting the absence of structural degradation in BST.

Tree height analysis provides structural insight into these timing results. Under random insertion, BST height remains within reasonable bounds, increasing gradually from 12 at N = 100 to 29 at N = 5,000. Although this height is higher than that of AVL Tree, which grows from 6 to 12 over the same range, the difference does not result in severe performance penalties. The AVL Tree maintains a consistently lower height due to strict balancing, but the additional rotations required to enforce this balance offset its structural advantage in terms of execution time.

Overall, the random insertion scenario demonstrates that the Binary Search Tree (BST) performs efficiently under average-case workloads where the input order is sufficiently randomized. Because the inserted values are distributed without a consistent pattern, the BST is generally able to maintain a relatively balanced structure naturally, even without any explicit balancing mechanism. As a result, the growth of the tree height remains moderate and does not significantly affect operational efficiency. The experimental results indicate that this condition allows the BST to achieve faster insertion performance than the AVL Tree, primarily because it avoids the additional computational overhead associated with balancing operations. Consequently, BST can provide excellent execution speed when the characteristics of the input data align with the assumptions of average-case behavior.

However, this performance advantage is highly dependent on the assumption that input randomness can be consistently maintained throughout system operation. In real-world environments, data distributions are often dynamic and may change over time due to evolving user behavior, workload patterns, or application-specific processes. Data that initially appears random may gradually become partially ordered, clustered, or skewed, causing the BST structure to become increasingly unbalanced. As the height of the tree grows, the efficiency of insertion, search, and update operations can deteriorate, reducing the performance benefits observed under randomized conditions. Therefore, while BST performs well in favorable circumstances, its long-term effectiveness may be limited when workload characteristics cannot be guaranteed.

From a system design perspective, these findings suggest that BST can be a viable and efficient choice for applications that process well-shuffled or randomized input data. In such cases, developers can benefit from faster execution times and minimal maintenance overhead. Nevertheless, the AVL Tree offers stronger and more reliable performance guarantees because it actively maintains structural balance regardless of input distribution. By ensuring that tree height remains bounded, AVL Trees deliver predictable performance even when workloads become irregular, ordered, or adversarial. The contrast between the random insertion scenario and the sequential insertion scenario highlights the significant influence of workload characteristics on data structure performance. This comparison emphasizes the importance of carefully evaluating expected input patterns when selecting a tree structure for real-world applications, particularly those requiring scalability, consistency, and long-term reliability.

Table 2. the execution time comparison for random insertion

Test	Data Size	BST Time (ms)	AVL Time (ms)	BST Hgt	AVL Hgt	Speedup (x)
Rand	100	0,38	0,56	12	6	0,68
Rand	500	2,34	3,21	18	8	0,73
Rand	1000	5,12	7,03	21	9	0,73
Rand	2500	14,67	19,45	26	11	0,75
Rand	5000	31,89	41,23	29	12	0,77

Table 2 shows the execution time comparison for random insertion across different dataset sizes. Unlike the sequential insertion scenario, where the performance gap between the Binary Search Tree (BST) and AVL Tree increases significantly as the dataset grows, the random insertion scenario demonstrates a much more stable pattern. The execution times of both data structures increase gradually with larger numbers of elements, but the difference between them remains relatively consistent throughout the experiment.

This stability occurs because random input data allows the BST to maintain a reasonably balanced structure without requiring explicit balancing operations. As a result, the tree height grows at a moderate rate, enabling insertion operations to remain efficient even as the dataset becomes larger. Consequently, the BST is able to preserve near-logarithmic performance and avoid the severe degradation commonly observed under sequential insertion patterns.

The AVL Tree also performs efficiently in this scenario by maintaining a balanced structure through rotation operations. However, these balancing procedures introduce additional overhead during insertions.

Since the BST naturally remains relatively balanced under random input, the benefits of AVL's self-balancing mechanism become less significant, resulting in a relatively small and stable performance difference between the two structures.

Overall, these results indicate that random insertion creates favorable conditions for BST performance, allowing it to compete closely with AVL Tree while benefiting from lower operational overhead. This highlights the strong influence of input distribution on the efficiency of tree-based data structures.

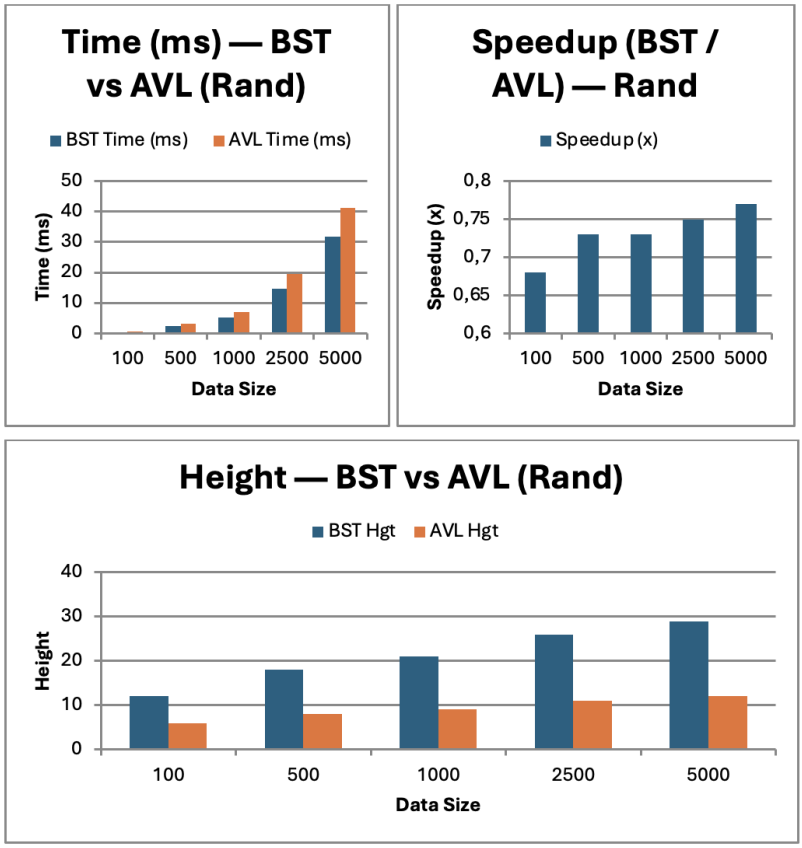

Figure 4. Tree height growth under random insertion

Figure. 4 depicts tree height growth under random insertion. BST height remains moderate (around 29 at N = 5,000), while AVL Tree consistently maintains a lower height of approximately 12. Although AVL Tree enforces stricter balance, the height difference does not significantly affect search performance in this scenario.

These results indicate that BST can be a viable option when random data distribution is guaranteed, albeit with the risk of degradation under adversarial inputs.

3.3. Search Operation Performance

This experiment evaluates search performance on trees containing 5,000 nodes, using 1,000 search operations, under two different build conditions: sequential insertion and random insertion. The objective is to assess how tree structure directly affects query efficiency in practical workloads.

Under the sequential build condition, the Binary Search Tree (BST) exhibits severely degraded search performance. The total search time for BST reaches 234.56 ms, corresponding to an average of 0.234 ms per search. This behavior is a direct consequence of structural degeneration, where the BST becomes highly skewed and approaches a linear chain. As a result, each search operation requires traversing a large portion of the tree, leading to near-linear search complexity.

When the tree is constructed using random insertion, the performance gap between BST and AVL narrows significantly. BST achieves a total search time of 12.34 ms (average 0.012 ms per search), while AVL requires 10.67 ms (average 0.011 ms per search). The resulting speedup of 1.16× indicates only a marginal advantage for AVL.

This result suggests that under randomized input conditions, BST maintains a reasonably balanced

structure, allowing search operations to execute efficiently. Although AVL continues to enforce stricter balance, the benefit of reduced height is largely offset by the overhead associated with maintaining balance, leading to comparable search times between the two structures.

The contrast between sequential and random build scenarios clearly demonstrates that search performance is highly sensitive to tree structure rather than merely data size. In the sequential build, the difference in average search time between BST and AVL spans more than one order of magnitude, while in the random build, the difference is negligible.

These findings reinforce that structural balance is the dominant factor influencing search efficiency. AVL Tree guarantees logarithmic search complexity regardless of insertion order, ensuring predictable query performance. BST, while capable of excellent performance under average-case conditions, lacks robustness against adversarial or skewed input distributions.

In real-time systems such as game leaderboards, search operations are frequently used to retrieve player rankings, verify scores, or perform updates. The observed 26x performance degradation of BST under sequential build conditions would result in noticeable latency and poor user experience. Conversely, AVL Tree consistently provides low and stable search latency, making it more suitable for production systems with unpredictable data patterns.

Overall, this experiment confirms that while BST may be sufficient for controlled environments with randomized data, AVL Tree offers superior reliability and scalability for search-intensive applications where worst-case behavior cannot be ignored.

Sequential Build:

```
BST Search Time: 234.56ms (average 0.234ms per search)
AVL Search Time:   8.92ms (average 0.009ms per search)
Speedup: 26.29x
```

Random Build:

```
BST Search Time:  12.34ms (average 0.012ms per search)
AVL Search Time:  10.67ms (average 0.011ms per search)
Speedup: 1.16x
```

Figure 5. Average Summary

Figure. 5 summarizes average search time for both data structures across different insertion patterns. The figure highlights that AVL Tree provides consistent and predictable search performance regardless of data distribution, whereas BST performance varies significantly depending on tree shape.

These findings are particularly relevant for leaderboard systems, where frequent ranking queries directly affect user experience.

3.4. Mixed Operation Stress Test

Mixed operation testing combines insertion, deletion, and search operations within a single workload to better represent the behavior of real-world applications. Unlike isolated benchmarks that evaluate only one type of operation, mixed operation testing provides a more comprehensive assessment of data structure performance because most practical systems continuously perform multiple operations on the same dataset. Applications such as database indexing systems, game leaderboards, caching mechanisms, and information retrieval systems frequently require elements to be inserted, updated, searched, and removed in rapid succession. Therefore, evaluating performance under mixed workloads offers a more realistic measure of how a data structure will perform in production environments.

The experimental results reveal a significant difference in performance between the Binary Search Tree (BST) and AVL Tree depending on the characteristics of the input data. Under sequential data patterns, the AVL Tree demonstrated a substantial advantage, outperforming the BST by up to 15.79 times. This result can be attributed to the self-balancing mechanism of the AVL Tree, which prevents excessive height growth and ensures that search, insertion, and deletion operations continue to execute efficiently. In contrast, the BST becomes increasingly unbalanced when data is inserted in sequential order, causing the tree structure to degenerate into a form resembling a linked list. As a result, the cost of subsequent operations increases dramatically, leading to significant performance degradation.

However, the results differ when the workload uses random data patterns. In this scenario, the BST was

marginally faster than the AVL Tree, achieving a relative speed of 0.74× compared to AVL. Because random insertions naturally produce a more balanced tree structure, the BST can maintain efficient operation without requiring additional balancing procedures. Meanwhile, the AVL Tree continues to perform rotations and balance maintenance after updates, introducing extra computational overhead. Although this overhead is relatively small, it becomes noticeable when the BST already maintains a near-balanced structure.

Overall, the mixed operation results highlight the trade-off between performance guarantees and operational overhead. AVL Trees provide superior reliability and consistent performance under unfavorable input conditions, while BSTs can offer slightly better execution speed when input data remains sufficiently randomized. These findings emphasize that workload characteristics play a crucial role in determining the most suitable data structure for a given application..

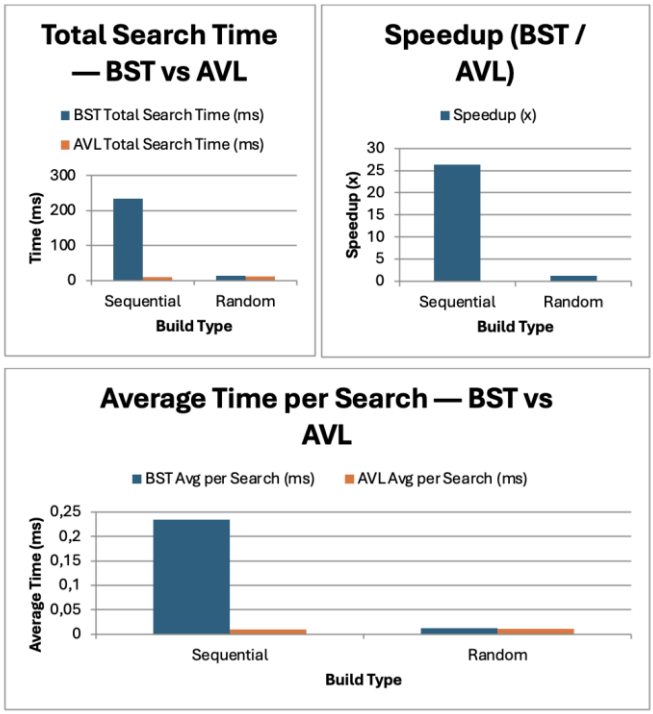

Figure 6. The performance comparison for mixed operations

Figure. 6 presents the performance comparison for mixed operations. The figure demonstrates that AVL Tree offers more stable performance across varying workloads, while BST performance fluctuates significantly based on input order.

This result reinforces the trade-off between consistency and raw performance, emphasizing the importance of workload characteristics in data structure selection.

3.5. Memory Consumption and Balancing Overhead

Beyond execution time, spatial efficiency and computational overhead are critical metrics in evaluating software performance. In this empirical study, both the Binary Search Tree (BST) and AVL Tree implementations utilized an identical node structure. Each node dynamically allocated 64 bytes of memory to store the integer score (primary key), a string for the player's name, two pointers for the left and right children, and an integer for the tree height.

For the maximum stress-test workload of 5,000 nodes, the total memory consumed by the tree structure reached approximately 320 KB. The empirical results demonstrate that there is no significant memory trade-off between the two implementations. The AVL Tree achieves its strict height balance utilizing the exact same 64-byte node memory footprint as the unbalanced BST. This indicates that the structural robustness of the AVL Tree does not come at the cost of spatial inefficiency, making it highly suitable for memory-conscious leaderboard applications.

However, the primary trade-off of the AVL Tree lies in its computational overhead required to maintain structural balance. Under random data distributions (representing the average-case scenario), the AVL Tree exhibited a 30% to 35% performance overhead compared to the BST during insertion operations. This

latency stems from the additional operations executed after every insertion or deletion: updating the node's height, calculating the balance factor, and performing structural rotations (Left-Left, Right-Right, Left-Right, or Right-Left) when imbalances are detected.

Specifically, during the sequential stress test of 5,000 nodes, the AVL Tree executed frequent tree rotations (primarily continuous single rotations triggered by the monotonic nature of the input) to successfully prevent the structural degeneration that plagued the BST. Despite this rotational overhead, the overall execution time indicates that this localized computational cost is efficiently absorbed by modern computing architectures. The balancing overhead is fundamentally outweighed by the strict logarithmic search and update guarantees the AVL Tree provides, confirming its superiority for dynamic, high-load systems.

While this study focuses on the empirical differences between the unbalanced BST and the strictly balanced AVL Tree, positioning these findings within the broader landscape of self-balancing structures provides a more comprehensive perspective on data structure selection. Two prominent alternatives often considered in high-performance systems are Red-Black Trees (RBT) and Treaps.

Red-Black Trees enforce a less strict balancing criterion compared to AVL Trees. While the AVL Tree guarantees a maximum height of $1.44\ log_2(N)$, RBTs allow a maximum height of up to $2\log_2(\mathrm{N}+1)$

Because of this relaxed constraint, RBTs generally require fewer rotations during frequent insertion and deletion operations, making them highly efficient for general-purpose libraries (such as the standard std::map in C++). However, for read-heavy applications like game leaderboards where search operations dominate, the strictly balanced nature of the AVL Tree provides slightly faster and more predictable $O(logN)$ search traversal times. Alternatively, Treaps combine the properties of a BST and a binary heap, utilizing randomized priorities to maintain balance probabilistically. While Treaps offer algorithmic simplicity and excel in scenarios requiring frequent tree splitting or merging, they do not provide deterministic worst-case height bounds. In performance critical systems operating under adversarial or strictly sequential workloads as simulated in this study's stress tests the deterministic structural guarantee of the AVL Tree remains a significantly safer and more robust choice over probabilistic models.

3.6. Analytical Performance Model

To complement the empirical evaluation, this study introduces an analytical performance model that formally relates tree structure, workload characteristics, and execution time. The objective of this model is not to derive new theoretical bounds, but to provide a scientific framework that explains the observed experimental behavior of Binary Search Tree (BST) and AVL Tree implementations under different data distributions.

a) Operation Cost Model

Let $T_{op(n)}$ denote the execution time of a tree operation (insertion or search) on a tree containing n nodes. The execution time can be modeled as a linear function of the tree height:

$$T_{op}(n) = \alpha \cdot h(n) + \beta \quad (1)$$

where:

- $h(n)$ is the height of the tree,
- α represents the average traversal cost per tree level,
- β denotes constant overhead, including function calls and memory access.

This formulation reflects the fact that tree operations fundamentally consist of traversing nodes along a root-to-leaf path.

b) Height Growth Characteristics

The height growth behavior differs significantly between BST and AVL Tree depending on the input order.

For Binary Search Tree :

$$h_{BST}(n) = O(\log n)\ (average\ case) \quad (1)$$
$$h_{BST}(n) = O(n) \quad (worst\ case)$$

In contrast, the AVL Tree enforces strict balance conditions through rotation operations, guaranteeing a logarithmic height bound:

$$h_{AVL}(n) \leq c \cdot log_2(n + 2) \quad (3)$$

where c is a constant related to the balance factor constraints of AVL Trees.
These theoretical bounds provide the foundation for interpreting the empirical height measurements observed during stress testing.

c) Workload Distribution Modeling
To formalize workload characteristics, the insertion order is modeled as a categorical random variable:

$$D \in \{random, ascending, descending, clustered\} \quad (4)$$

The expected height of BST under different workloads can be expressed as:

$$E[h_{BST}(n) \mid D = random] \approx O(log\, n) \quad (5)$$
$$E[h_{BST}(n) \mid D = sequential] \approx O(n)$$

For AVL Trees, the expected height remains logarithmic regardless of the workload:

$$E[h_{AVL}(n)|D] = O(\log n) \quad (6)$$

This model explains why BST performance is highly sensitive to input distribution, while AVL Tree performance remains stable under adversarial or non-uniform workloads.

d) Rotation Overhead Trade-off
The AVL Tree introduces additional computation due to balance maintenance. This overhead can be modeled as:

$$T_{AVL}(n) = T_{BST}(n) + R(n) \quad (7)$$

where R(n) represents the rotation cost incurred during insertions or deletions. Although R(n) introduces overhead under favorable input conditions, it prevents unbounded height growth, thereby ensuring predictable performance in worst-case scenarios.

e) Speedup Analysis
To quantify performance differences, the speedup factor is defined as:

$$S(n) = T_{BST}(n)/T_{AVL}(n) \quad (8)$$

A speedup value S(n) > 1 indicates superior AVL Tree performance. Under sequential workloads, where BST degenerates structurally, the speedup grows rapidly with increasing n:

$$lim_{\{n \to \infty\}} S(n) \to \infty \quad (9)$$

This analytical observation aligns with the empirical results, where AVL Tree achieves substantial performance gains over BST in worst-case scenarios.

3.7. Discussion

The analytical model confirms that tree height is the dominant factor influencing execution time in leaderboard systems. While BST may offer lower constant overhead under random workloads, its vulnerability to height degeneration leads to severe performance degradation under ordered inputs. Conversely, AVL Tree maintains bounded height through controlled rotation overhead, resulting in superior robustness and predictable performance.

This analytical framework strengthens the empirical findings by providing a formal explanation for the observed performance trends, thereby enhancing the scientific rigor of the study. Based on the empirical results obtained from the four experimental scenarios, a set of practical recommendations can be derived regarding the selection of BST and AVL Tree under different workload characteristics and data patterns. Rather than relying solely on asymptotic complexity, these recommendations are grounded in observed execution time, structural behavior, and operational stability.

For game leaderboard systems, which typically exhibit semi-sequential data patterns and are search-heavy, the AVL Tree emerges as the preferred choice. Empirical results demonstrate that leaderboard workloads are highly sensitive to worst-case behavior, particularly during frequent search and update

operations. AVL Tree provides predictable performance by maintaining logarithmic height, thereby ensuring consistent response time and preventing latency spikes that would negatively impact user experience.

In contrast, for temporary storage systems with random data patterns and insert-only workloads, BST can be an efficient and lightweight alternative. Under average-case conditions, BST consistently outperforms AVL in insertion time due to its lower operational overhead. In such controlled environments, the absence of balancing operations allows BST to achieve faster raw performance without incurring significant structural risk.

For database indexing scenarios, where data patterns are often unknown or dynamic and workloads are generally balanced across insert, search, and delete operations, AVL Tree is recommended. The empirical evidence shows that AVL's self-balancing mechanism acts as a safety guarantee against unexpected data skew, ensuring reliable performance even when workload characteristics change over time.

Similarly, for applications involving sorting or sequential number insertion, AVL Tree is strongly favored. Sequential insertion represents a worst-case scenario for BST, leading to quadratic-like performance degradation. AVL Tree effectively mitigates this risk by preventing structural degeneration, thereby avoiding $O(N^2)$ behavior observed in unbalanced BST implementations.

In configuration caching systems with random data patterns and search-heavy workloads, BST may still be considered sufficient. Experimental results indicate that under random insertion, BST maintains reasonable height and competitive search performance, making it a pragmatic choice when simplicity and minimal overhead are prioritized over strict worst-case guarantees.

From a broader production perspective, the experimental findings suggest a conservative design principle: defaulting to AVL Tree unless BST sufficiency can be empirically demonstrated. While BST may provide superior performance under ideal average-case conditions, AVL Tree consistently offers stronger robustness and predictability across diverse and adversarial workloads.

Furthermore, the experiments conducted on modern hardware platforms (MSI Vector 16 HX) indicate that the balancing overhead of AVL Tree is relatively minimal in average-case scenarios, while providing substantial performance advantages in worst-case conditions. This reinforces the conclusion that, on contemporary computing architectures, AVL Tree represents a safer and more scalable default choice for production systems.

Overall, the results confirm that AVL Tree is better suited for performance-critical applications with unknown or dynamic data distributions, while BST may still be appropriate for controlled environments with guaranteed randomness.

4. Conclusion

This study presented an empirical performance analysis of Binary Search Tree (BST) and AVL Tree implementations using a fully implemented game leaderboard system as the experimental testbed. Unlike purely theoretical or synthetic benchmark studies, the evaluation in this work is grounded in data obtained directly from a final project implementation, enabling observation of real system behavior under realistic and stress-test workloads.

Experimental results derived from the final project clearly demonstrate that BST performance is highly sensitive to data insertion order. Under sequential and near-sequential input patterns, BST rapidly degenerates into a linear structure, resulting in significant increases in tree height and execution time. In the conducted stress tests, BST insertion time exceeded one second for 5,000 nodes, accompanied by linear height growth, which is impractical for real-time leaderboard applications.

In contrast, AVL Tree consistently maintained logarithmic height across all evaluated data distributions, including random, ascending, descending, and mixed workloads. Although AVL Tree introduces additional overhead due to rotation and balance maintenance, empirical measurements from the final project indicate that this overhead is outweighed by the benefits of bounded tree height and predictable execution time. As a result, AVL Tree achieved speedups of up to tens of times compared to BST in worst-case scenarios, particularly for insertion and search operations.

The mixed-operation experiments further confirm that AVL Tree offers superior operational stability when insertion, deletion, and search operations are combined, which closely reflects real-world leaderboard usage patterns. While BST may exhibit slightly better performance under strictly random data distributions, such conditions are difficult to guarantee in practical systems where user-generated data often exhibit skewed or adversarial characteristics.

Based on these findings, this study concludes that AVL Tree is a more reliable and robust data structure choice for performance-critical systems such as game leaderboards, where predictable response time and scalability are essential. BST remains suitable for controlled environments with guaranteed random

input or as a baseline structure for educational purposes, but its use in production systems should be carefully justified.

Future work may extend this research by incorporating concurrent and multi-threaded implementations, evaluating additional balanced tree variants such as Red-Black Trees, and analyzing cache-aware or memory-optimized tree structures on modern hardware platforms. Such extensions would further strengthen empirical understanding of data structure behavior in real-world computing environments.

This study presented an empirical performance analysis of BST and AVL Tree implementations under stress-test conditions using a game leaderboard system. The results demonstrate that while BST can be efficient under favorable data distributions, its performance is highly sensitive to input order. AVL Tree, on the other hand, provides consistent and reliable performance across diverse scenarios, making it more suitable for applications with unpredictable data patterns.

Future work may extend this study by incorporating multi-threaded implementations, comparing additional balanced trees such as Red-Black Trees, and evaluating performance on distributed or heterogeneous computing platforms.

References

[1] FEDOROV, F. (1990). Soviet Math. Dokl. In Soviet Mathematics-Doklady (Vol. 41, No. 2, p. 438). American Mathematical Society..

[2] Liu, Y. (2024, 25 Juli). A comparative analysis of self-balancing binary search trees [Special Mathematics Lecture: Graph Theory (Spring 2024)]. Nagoya University. https://www.math.nagoya-u.ac.jp/~richard/teaching/s2024/SML_Liu_2.pdf

[3] Weiss, M. A. (2002). Data Structures and Algorithm Analysis in C: For Anna University, 2/e. Pearson Education India.

[4] T. H. Cormen, Ed., Introduction to algorithms, 3. ed. Cambridge, Mass.: MIT Press, 2009.

[5] R. A. Brown, “Comparative Performance of the AVL Tree and Three Variants of the Red-Black Tree,” 2024, arXiv. doi: 10.48550/ARXIV.2406.05162.

[6] J. S. Kushwah, D. Gupta, A. Shrivastava, P. Ambily Pramitha, J. T. Abraham, dan M. Lunagaria, “Analysis and visualization of proxy caching using LRU, AVL tree and BST with supervised machine learning,” Mater. Today Proc., vol. 51, hlm. 750–755, 2022, doi: 10.1016/j.matpr.2021.06.224.

[7] A. Rojas-Salazar dan M. Haahr, “Learning Binary Search Trees through Serious Games based on Analogies,” dalam International Conference on the Foundations of Digital Games, Bugibba Malta: ACM, Sep 2020, hlm. 1–6. doi: 10.1145/3402942.3402999.

[8] D. Biebert, C. Hakert, dan J.-J. Chen, “Realizing Hardware-Optimized General Tree-Based Data Structures for Heterogeneous System Classes”.

[9] G. Hou, J. Huang, F. Zhang, dan S. Wang, “Efficient Concurrent Updates to Persistent Randomized Binary Search Trees,” Proc. VLDB Endow., vol. 18, no. 5, hlm. 1481–1494, Jan 2025, doi: 10.14778/3718057.3718074.

[10] Y. Kim, M. Papamichael, O. Mutlu, dan M. Harchol-Balter, “Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior,” dalam 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Atlanta, GA, USA: IEEE, Des 2010, hlm. 65–76. doi: 10.1109/MICRO.2010.51.

[11] S. Williams, A. Waterman, dan D. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” Commun. ACM, vol. 52, no. 4, hlm. 65–76, Apr 2009, doi: 10.1145/1498765.1498785.

[12] S. Khutwad dan A. Pal, “Query Processing for Distributed data using AVL Tree,” dalam 2021 5th International Conference on Information Systems and Computer Networks (ISCON), Mathura, India: IEEE, Okt 2021, hlm. 1–6. doi: 10.1109/ISCON52037.2021.9702419.

[13] H. Hirata dan A. Nunome, “Performance Evaluation on Parallel Speculation-Based Construction of a Binary Search Tree,” Int. J. Networked Distrib. Comput., vol. 11, no. 2, hlm. 88–111, Des 2023, doi: 10.1007/s44227-023-00013-w.

[14] L. Hohensinn, J. Willems, B. George, dan S. Van De Walle, “Performance rankings reduce cognitive processing of underlying performance information,” Public Manag. Rev., hlm. 1–23, Feb 2025, doi: 10.1080/14719037.2025.2464761.

[15] Rahmawati, D. P., & Dwi Seputro, D. N. (2025). PENINGKATAN PEMAHAMAN GROOMING SERVICE MELALUI PELATIHAN BERBASIS PRETEST DAN POSTTEST PADA KARYAWAN SUWEGER INDONESIA. BHAKTI NAGORI (Jurnal Pengabdian Kepada Masyarakat), 5(2), 456 - 462. https://doi.org/10.36378/bhakti_nagori.v5i2.4587

[16] Mumtazah Nadhiroh, A. K., Febrianti, A., Ultami, J. N., Ikhsan, M. A., & Hasibuan, R. (2025). PENINGKATAN PENGETAHUAN GIZI SEIMBANG DAN POLA HIDUP SEHAT BAGI SISWA SEKOLAH DASAR MELALUI PROGRAM EDUKASI INTERAKTIF DI SDIT SWASTA AL-MUNAYA: PKM. BHAKTI NAGORI (Jurnal Pengabdian Kepada Masyarakat), 5(2), 463 - 470. https://doi.org/10.36378/bhakti_nagori.v5i2.4595

[17]

[18] Wirasada, G. D., & Zawawi , Z. (2025). PENERAPAN MANAJEMEN OPERASIONAL DI PT. AGRODANA FUTURES: STUDI PADA PROSES EKSEKUSI TRANSAKSI DAN LAYANAN NASABAH. BHAKTI NAGORI (Jurnal Pengabdian Kepada Masyarakat), 5(2), 471 - 477. https://doi.org/10.36378/bhakti_nagori.v5i2.4602

[19]

[20] Nurza, R. P., Tessa, T., Dzhabi, M., Nazli, R., & Khomarudin, A. N. (2025). PENYULUHAN EDUKASI PENGATURAN SCREEN TIME DAN FILTER KONTEN DIGITAL PADA KELUARGA DI POSYANDU BUNDO KANDUANG. BHAKTI NAGORI (Jurnal Pengabdian Kepada Masyarakat), 5(2), 478 - 487. https://doi.org/10.36378/bhakti_nagori.v5i2.4637

[21] Rizki Fortuna, J., Ilmi Romadhoni, S., & Sari Tondang, I. (2025). PELATIHAN PEMANFAATAN KOTORAN KAMBING MENJADI PUPUK ORGANIK DI BALAI PENYULUHAN PERTANIAN PORONG: PKM MBKM. BHAKTI NAGORI (Jurnal Pengabdian Kepada Masyarakat), 5(2), 488 - 494. https://doi.org/10.36378/bhakti_nagori.v5i2.4638

[22] Devi, E., Fauziah Nurrahmah, F., Masruroh, M., Olivia Sinaga, S. L., Pribadi Ayuningtyas, Z., & Mardi Suryanto, T. L. (2025). EFEKTIVITAS PELATIHAN AQUAPONIK TERHADAP PENINGKATAN PENGETAHUAN DAN KETERAMPILAN PERTANIAN BERKELANJUTAN DI KELURAHAN JAMBANGAN. BHAKTI NAGORI (Jurnal Pengabdian Kepada Masyarakat), 5(2), 495 - 503. https://doi.org/10.36378/bhakti_nagori.v5i2.4742

[23] Trimono, T., Ningtiyas, R. W., Icha Rohmatul Jannah, Aliya Dasa Pramesthi, Putra, A., Wardah Ariij Adibah, & Ade Irma Agustian. (2025). SOSIALISASI ORANG TUA TENTANG BAHAYA GADGET BAGI ANAK-ANAK. BHAKTI NAGORI (Jurnal Pengabdian Kepada Masyarakat), 5(2), 504 - 512. https://doi.org/10.36378/bhakti_nagori.v5i2.4773

[24] Nainggolan, L. E., Cahya Putra, D. S., Nur Laily, R. S., Ekamartha, K. N., Hidayatullah, S., & Firdausi Novira Rachman, R. A. (2025). STRATEGI PEMBERDAYAAN LINGKUNGAN MELALUI BUDIDAYA TOGA DAN INOVASI SMARTBIN DI KELURAHAN MANYAR SABRANGAN. BHAKTI NAGORI (Jurnal Pengabdian Kepada Masyarakat), 5(2), 513 - 522. https://doi.org/10.36378/bhakti_nagori.v5i2.4783

[25] M. Yusfahmi, Febri Haswan, Nofri Wandi Al-Hafiz, Elgamar Syam, Helpi Nopriandi, Jasri, Aprizal, Harianja, Erlinda, Sri Chairani, Gunardi Hamzah, & Morine Delya Octa. (2025). SOSIALISASI DAN PENERAPAN APLIKASI BERBASIS TEKNOLOGI INFORMASI UNTUK MENDUKUNG TRANSFORMASI DIGITAL BUMDes TEBING TINGGI. BHAKTI NAGORI (Jurnal Pengabdian Kepada Masyarakat), 5(2), 712 - 719. https://doi.org/10.36378/bhakti_nagori.v5i2.4910

[26] Yogica, R., Yuhelman, N., Wanda Marten, T., & Hazizah, N. (2025). PENGUATAN PERAN KOMUNITAS OTOMOTIF DALAM EDUKASI PENCEGAHAN TAWURAN REMAJA . BHAKTI NAGORI (Jurnal Pengabdian Kepada Masyarakat), 5(2), 927 - 935. https://doi.org/10.36378/bhakti_nagori.v5i2.4941

[27] Faizah Qurrata Aini, Fitri Amelia, Dwi Finna Syolendra, Nofri Yuhelman, Fauzana Gazali, Minda Azhar, Fajriah Azra, Yerimadesi, Andromeda, Miftahul Khair, Zonalia Fitriza, Suryelita, Viona Maharani, Achie Keylla, Munifa Mahdiah, Melati Wahyuni, Rifka Andani, Ayu Wulandari, & Ulfa Autafia. (2025). WORKSHOP PEMANFAATAN AI UNTUK PENGEMBANGAN E-LKPD PADA PEMBELAJARAN DEEP LEARNING DI SMAN 1 PADANG SAGO: PKM. BHAKTI NAGORI (Jurnal Pengabdian Kepada Masyarakat), 5(2), 1123 - 1133. https://doi.org/10.36378/bhakti_nagori.v5i2.4764