TOPIC 3: Performance comparison of an AVL tree and B-tree

1. Introduction

Data structures play a central role in the performance of modern software systems, particularly when handling large collections of data that require efficient insertion, search, and deletion operations. Balanced search trees are widely used to guarantee logarithmic time complexity for these operations, regardless of the order of the input data.

Among balanced tree structures, AVL trees and B-Trees represent two different design philosophies. AVL trees enforce a strict height-balance constraint, ensuring that the difference in height between the left and right subtrees of any node is at most one. This property guarantees fast search operations but may introduce additional overhead due to frequent rebalancing rotations during insertions and deletions.

B-Trees, on the other hand, are multi-way search trees designed to minimize tree height by allowing each node to store multiple keys. Originally developed for disk-based storage systems, B-Trees reduce the number of memory accesses by increasing node capacity, which makes them particularly efficient when working with large datasets or hierarchical memory architectures.

The objective of this seminar is to experimentally compare the performance of AVL trees and B-Trees under different configurations. We evaluate their behavior in terms of execution time and memory usage for insertion, search, and deletion operations, using datasets of increasing size. Special attention is given to the impact of the B-Tree order on performance, as well as to the alignment between theoretical complexity and empirical results.

The complete source code and experimental setup are publicly available on GitHub:
https://github.com/ChartonMatthieu/avl_vs_btree

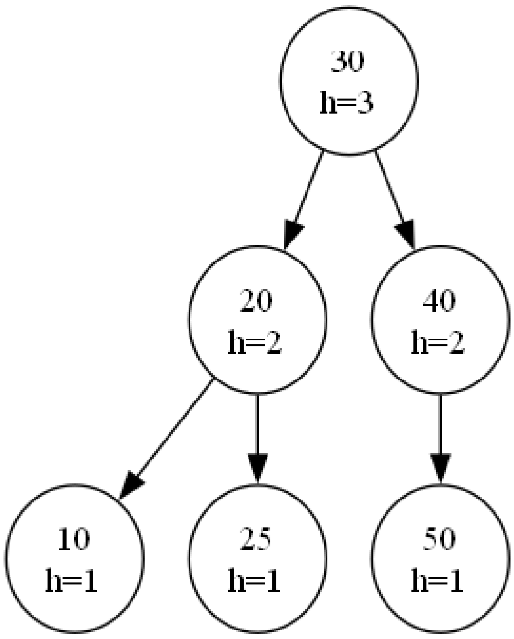
2. Algorithm Presentation

2.1 AVL Tree

An AVL tree is a self-balancing binary search tree in which the height of the left and right subtrees of any node differs by at most one. Each node stores a height value, which allows the balance factor to be computed efficiently after insertion or deletion operations.

When an insertion or deletion causes the balance factor of a node to violate the AVL constraint, the tree is rebalanced using rotations. Four cases may occur: left-left (LL), right-right (RR), left-right (LR), and right-left (RL). These rotations restore the height balance locally while preserving the binary search tree property.

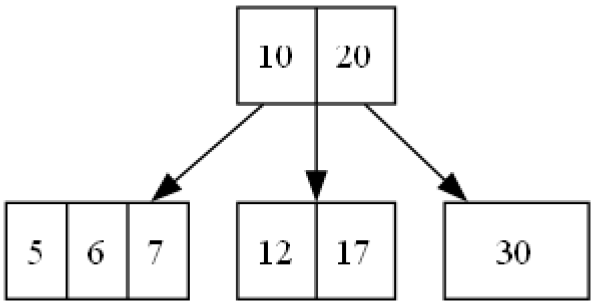
Due to its strict balancing rules, the AVL tree guarantees a height of O(log n), which ensures logarithmic time complexity for search, insertion, and deletion operations. However, this strictness may lead to a higher number of rotations compared to other balanced tree structures, increasing the constant factors in practice.



Graphviz figure of an AVL

2.2 B-Tree

A B-Tree is a balanced multi-way search tree parameterized by its minimum degree t, which determines the maximum and minimum number of keys stored in each node. A node may contain up to 2t−1 keys and, except for the root, at least t−1 keys. All leaves are located at the same depth, ensuring a balanced structure.
Unlike binary search trees, B-Trees reduce their height by storing multiple keys per node. During insertion, if a node becomes full, it is split and its median key is promoted to the parent node. This splitting process may propagate up to the root, potentially increasing the height of the tree.
In this work, B-Trees were implemented with different values to study the impact of node capacity on performance. Larger values of t reduce the height of the tree but increase the size of each node, leading to a trade-off between memory usage and execution time.
While B-Trees are particularly well-suited for external memory systems, they also offer interesting performance characteristics in main memory due to improved cache locality.



Graphviz figure of a B-Trees

3. Experiment Setup

3.1 Experimental Environment

All experiments were conducted on a standard personal computer using a single-core execution model to avoid interference from parallelism. The implementation was developed in Python 3, using only standard libraries and well-known third-party packages for profiling and visualization.

Execution time measurements were performed using the time.perf_counter() function, which provides high-resolution wall-clock timing suitable for benchmarking. Memory usage was measured using the tracemalloc module, allowing us to track peak memory consumption during insertion operations.

To ensure reproducibility, all experiments were executed using a fixed random seed. The complete source code, datasets, and instructions required to reproduce the experiments are publicly available in a GitHub repository.

3.2 Data Structures and Configurations

Two balanced tree data structures were evaluated: an AVL tree and a B-Tree. The AVL tree implementation follows the classical design, storing one key per node, and maintaining balance through rotations after insertions and deletions.

For the B-Tree, several configurations were tested by varying the minimum degree parameter t. Specifically, B-Trees with t = 2, t = 3, and t = 5 were implemented, corresponding to maximum node capacities of 3, 5, and 9 keys respectively. This choice allows us to study the impact of node capacity on performance and memory usage.

All tree implementations support insertion and search operations. Deletion was fully implemented and evaluated for the AVL tree, while deletion in B-Trees was not benchmarked due to its higher algorithmic complexity and the focus of this study on insertion and search behavior

3.3 Datasets

The experiments were performed on datasets composed of integer values only, which is sufficient to analyze the structural and performance characteristics of the data structures under study.

For each experiment, datasets of increasing size were generated to observe scalability trends. The following dataset sizes were used: 1,000, 5,000, 10,000, and 50,000 elements. Values were drawn randomly without replacement from a bounded integer range to avoid duplicates and ensure consistent tree growth behavior.

Using incremental dataset sizes allows for a clear comparison of asymptotic trends and highlights differences in performance as the number of stored elements increases.

3.4 Experimental Procedure

For each dataset size, a fresh instance of each data structure was created to avoid contamination between runs. Insertions were performed sequentially for all values in the dataset, and the total insertion time was recorded.

After the insertion phase, search operations were executed for all inserted values to measure lookup performance under optimal conditions. Memory usage was measured during the insertion phase, as it represents the point at which the data structure reaches its maximum size.

Each experiment was repeated for all B-Tree configurations and for the AVL tree, ensuring a fair comparison under identical conditions. Results were collected and stored for further analysis and visualization.

3.5 Visualization and Validation

To validate the correctness of the implementations, both AVL trees and B-Trees were visualized using Graphviz. These visualizations allowed us to verify structural properties such as height balance in AVL trees and uniform leaf depth in B-Trees.

Visual inspection of the trees after insertion and deletion operations confirmed that all structural invariants were preserved throughout the experiments.

Performance graphs were generated automatically using Matplotlib during the benchmark execution.

4. Experiment Results

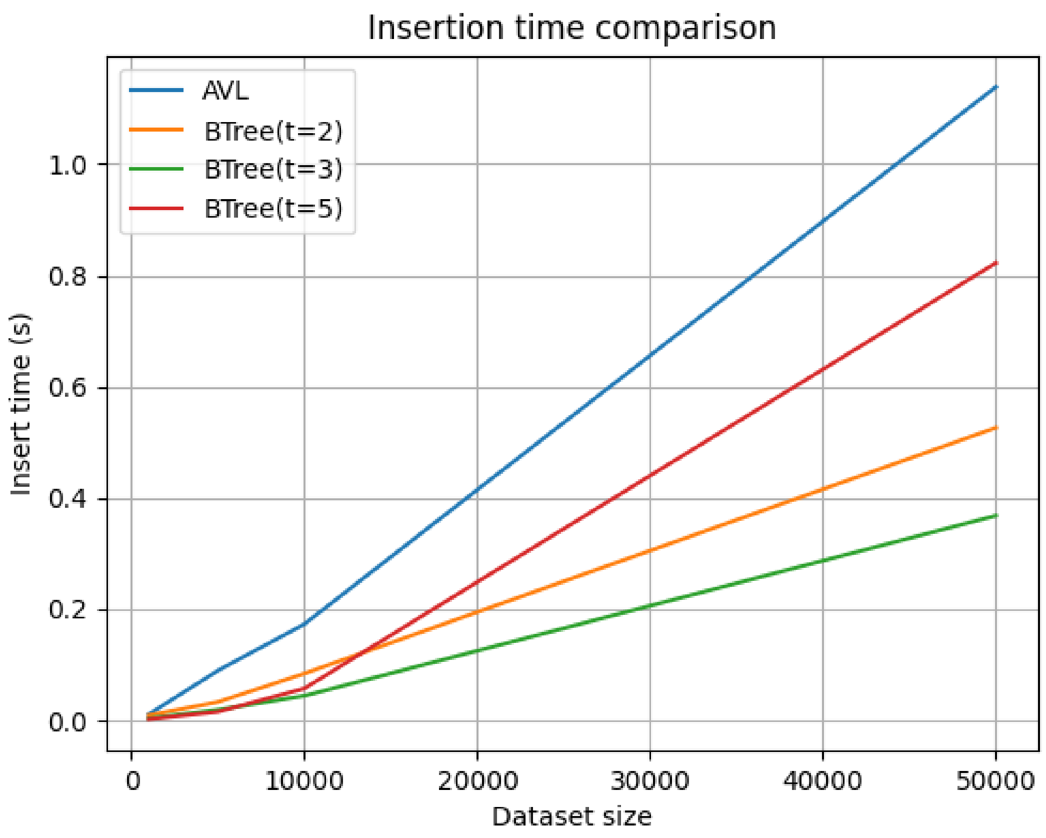
4.1 Insertion Time

Figure below presents the comparison of insertion time as a function of the dataset size for the AVL tree and the different B-Tree configurations. For all tested structures, insertion time increases with the number of elements, following an approximately linear trend on the tested range.

The AVL tree consistently exhibits the highest insertion time across all dataset sizes. This behavior is expected due to the strict balancing constraints of AVL trees, which require frequent rotations to maintain a height difference of at most one between subtrees.

In contrast, all B-Tree configurations outperform the AVL tree in terms of insertion time. Among them, higher values of the minimum degree t led to better performance. In particular, the B-Tree with t = 5 shows the lowest insertion time, followed by t = 3 and t = 2. This improvement can be explained by the reduced height of higher-order B-Trees, which decreases the number of node traversals and rebalancing operations required during insertion.

Overall, these results confirm the theoretical expectation that B-Trees benefit from increased node capacity, allowing them to achieve better insertion performance at the cost of larger nodes.



4.2 Search Time

Search operations were performed after the insertion phase for all values present in the dataset. Across all experiments, search times remained significantly lower than insertion times and showed less variation between data structures.

The AVL tree and B-Trees demonstrated comparable search performance, as all structures maintain logarithmic height. However, B-Trees with higher values of t tend to exhibit slightly better performance due to their reduced height, which lowers the number of comparisons required during traversal.

Since search operations do not trigger rebalancing, performance differences are primarily influenced by tree height rather than structural maintenance costs.

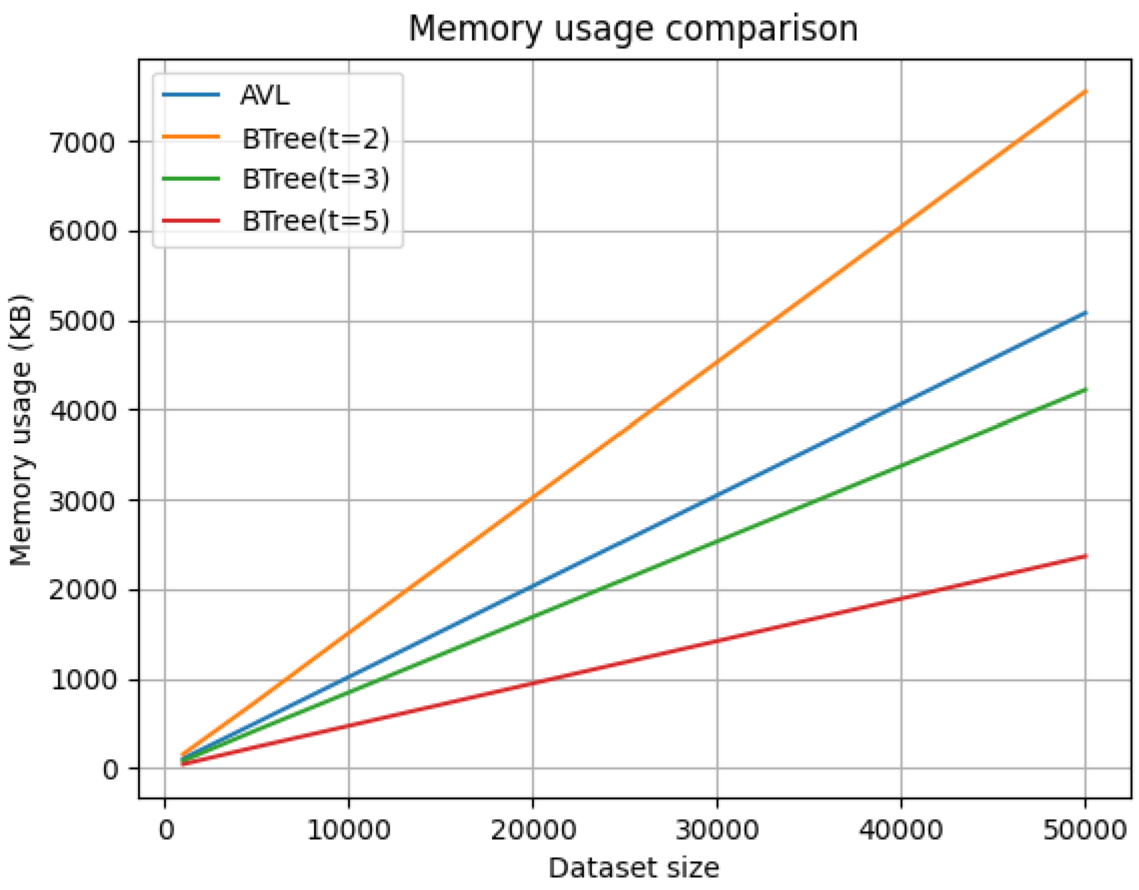
4.3 Memory Usage

Figure below illustrates the memory usage of each data structure as a function of the dataset size. As expected, memory consumption increases linearly with the number of stored elements for all configurations.

The B-Tree with t = 2 exhibits the highest memory usage among all tested structures. This result can be attributed to the large number of nodes required when node capacity is small, leading to higher pointer and structural overhead.

Interestingly, B-Trees with higher minimum degrees show a significant reduction in memory usage. In particular, the B-Tree with t = 5 consumes less memory than both the AVL tree and the lower-order B-Trees. This behavior is explained by the fact that higher-

order B-Trees store more keys per node, reducing the total number of nodes and amortizing structural overhead.
Although B-Trees are often considered memory-intensive, these results demonstrate that higher-order B-Trees can be more memory-efficient than AVL trees in practice when storing large datasets in main memory.



4.4 Summary of Results

The experimental results highlight clear differences between AVL trees and B-Trees. AVL trees incur higher insertion costs due to strict balancing requirements, while B-Trees achieve better performance by reducing tree height through increased node capacity.
The choice of the B-Tree parameter has a strong impact on both execution time and memory usage. Higher values of t lead to faster insertion times and lower memory consumption, at the expense of larger node sizes.
Overall, the observed trends are consistent with theoretical complexity analyses and confirm the advantages of B-Trees for handling large datasets efficiently.

5. Discussion and Conclusion

5.1 Discussion

The experimental results obtained in this study largely align with the theoretical expectations regarding balanced tree data structures. AVL trees, which enforce strict

height balance, provide reliable logarithmic performance for all operations but incur additional costs due to frequent rotations during insertions and deletions.
B-Trees, by contrast, demonstrate superior insertion performance as their order increases. By allowing multiple keys per node, B-Trees significantly reduce tree height, which leads to fewer node traversals and improved cache locality. This advantage becomes increasingly evident as the dataset size grows.
Memory usage results also highlight an important trade-off. While B-Trees are often perceived as memory-heavy structures, our experiments show that higher-order B-Trees can be more memory-efficient than AVL trees. This behavior is explained by the reduced number of nodes required to store the same number of keys, which amortizes pointer and structural overhead.
It is important to note that this study focuses on in-memory data structures implemented in Python. While B-Trees were originally designed for external memory systems, their performance benefits still translate to main memory due to reduced tree height and improved cache behavior. However, the absolute performance values are influenced by Python's object model and memory management, which may differ from lower-level implementations in languages such as C or C++.

5.2 Limitations

Several limitations should be acknowledged. First, deletion operations were benchmarked only for AVL trees. Deletion in B-Trees involves complex rebalancing operations, including key redistribution and node merging, which were beyond the primary scope of this study.
Second, the experiments were conducted using randomly generated datasets. Although this approach is suitable for analyzing average-case behavior, different input distributions such as sorted or nearly sorted data could further influence performance.
Finally, all experiments were executed in a single-threaded environment. While this choice ensures reproducibility, it does not account for potential parallel optimizations that could affect real-world applications.

5.3 Conclusion

This seminar presented an experimental comparison between AVL trees and B-Trees under varying configurations. By evaluating insertion time, search performance, and memory usage on datasets of increasing size, we highlighted the practical trade-offs between these two balanced tree structures.
The results demonstrate that AVL trees provide predictable performance but suffer from higher insertion costs due to strict balancing. B-Trees, especially those with higher minimum degrees, offer superior insertion performance and improved memory efficiency by reducing tree height and node count.
Overall, the findings confirm that B-Trees are well-suited for managing large datasets where insertion performance and memory locality are critical, while AVL trees remain a robust choice for scenarios requiring strict balance guarantees. Future work could

extend this study by evaluating deletion performance in B-Trees, exploring additional dataset distributions, or implementing the data structures in lower-level languages to reduce runtime overhead.