COMP0005 Group Coursework
Academic Year 2024-25

Group Number: 10

1 Overview of Experimental Framework

1.1 Framework Design/Architecture

The experimental framework is designed to evaluate the performance of three search data structures, AVL Tree, LLRB BST, and Scapegoat Tree, for string-based insertion and search operations under four dataset conditions: unsorted, sorted, partially sorted (10% swaps), and a dataset with uppercase letters and digits for non-existent element searches. The first three assess insertion performance, while the fourth tests worst-case search efficiency. All datasets contain 1,000,000 unique 5-character strings.

The evaluation process adopts incremental testing, inserting elements into the same tree in batches rather than all at once, allowing for a progressive tree construction. By recording the number of elements at insertion (as the x-axis) and the average insertion time per operation (as the y-axis), this method provides a more precise measure of individual insertion costs, revealing performance variations due to tree growth, such as rebalancing overhead. Compared to the total time averaging method, this approach offers finer-grained insights.

In a randomized insertion scenario, a search test is conducted immediately after each insertion, querying 100 sampled elements to simulate real-world scenarios where insertions and searches occur concurrently. The test includes both present and absent elements: successful lookups evaluate retrieval efficiency, exposing balancing differences, while failed lookups assess worst-case complexity, reflecting tree depth and unsuccessful search costs—critical for practical applications.

The evaluation uses dynamic sampling, recording times at 100 logarithmically distributed points, with dense sampling for the first 99 points (1, 3, ..., 99) and logarithmic scaling up to 1,000,000 thereafter. This enables multi-scale analysis while reducing evaluation overhead and ensuring sufficient data for smooth curve plotting.

To mitigate system noise, repeated testing and adaptive smoothing strategies are applied. The former reduces random fluctuations by executing identical operations multiple times, while the latter dynamically adjusts smoothing windows based on variance, preventing excessive smoothing (which may obscure real trends) or insufficient smoothing (which retains excessive noise).

1.2 Hardware/Software Setup for Experimentation

Experiments were conducted on a local machine with an Intel Core i9-14900HX processor (24 cores, 2.2GHz base frequency, around 4.24GHz during execution), 32 GB DDR5 RAM, and a 1 TB NVMe SSD.

The software environment consisted of Windows 11 Professional (Version 23H2), running Python 3.11.4 was executed within Visual Studio Code using the Jupyter Notebook extension, which leveraged the Python interpreter selected via the VS Code Python extension. Only permitted libraries were used: timeit, random, string, math, and matplotlib. Experiments ran locally on a single thread, with non-essential background processes (e.g., browsers, antivirus) disabled to minimize system noise. This setup, combined with the framework’s noise-mitigation techniques (e.g., repeated measurements, smoothing), ensured that performance differences reflected algorithmic efficiency, enabling a fair and accurate comparison.

2 Performance Results

2.1 AVL Tree

Evaluation results show that AVL trees exhibit consistent performance across different input sequences (random, sorted, and nearly sorted), with insertion times demonstrating a steady logarithmic growth pattern. AVL trees efficiently handle imbalance caused by ordered inputs through localized rotations (O(1)), mitigating structural degradation despite frequent rotations. Theoretically, AVL insertions involve balance factor checks and rotational adjustments. While ordered insertions cause imbalance by continuously adding nodes to one subtree (e.g., the right), the localized nature and constant-time complexity of each rotation distribute the cumulative cost to O (log n). In contrast, random insertions introduce dispersed imbalances, requiring more height checks or traversals, incurring hidden overhead.

Furthermore, ordered inputs improve cache locality, as insertions are concentrated in a specific region (e.g., the right subtree), enhancing cache hit rates and reducing memory overhead. Conversely, unordered insertions lead to scattered node accesses, increasing cache misses and branch mispredictions. In large datasets, the locality advantage of ordered inputs offsets rotation costs, making insertion times comparable or even lower than those of random inputs. As data scales, logarithmic height dominance further diminishes the impact of constant rotation costs. Combined with rotation optimizations and cache effects, ordered inputs can achieve equal or superior performance, demonstrating AVL trees' adaptability and efficiency (Appendix C).

AVL trees maintain high efficiency, stability, and similar performance levels for both existing and non-existing elements due to their strict height balance. Regardless of key presence, search path length remains bounded by tree height O(log n), ensuring clear, redundancy-free traversal. Successful searches terminate at target nodes, while failed searches reach leaf nodes, resulting in similar depths and minimal comparison differences, highlighting AVL trees’ symmetric efficiency and reliability.

2.2 LLRBST

For random input, the LLRB tree demonstrates a stable increase in insertion time as the number of elements grows. The balancing operations (rotations and colour flips) effectively maintain an O(log n) cost, which is evident from the gradual rise in the time required to insert new elements. The curve obtained from the experiment shows that the LLRB tree efficiently handles randomness in the input while keeping rebalancing overhead under control.

When data is inserted in a strictly sorted order, the LLRB tree actively rebalances itself to avoid the formation of a degenerate (skewed) structure. The measured insertion times indicate that while the tree does perform additional rebalancing steps, the overall cost per insertion remains efficient. The performance trend confirms that even under sorted input conditions, the tree's balancing mechanisms prevent significant degradation in insertion speed.

With almost sorted data, the LLRB tree faces fewer imbalances compared to the strictly sorted case. This results in slightly lower rebalancing overhead, which is reflected in a modest improvement in the insertion times. The performance graph for this scenario shows that the LLRB tree efficiently corrects minor discrepancies in order, thereby maintaining a balanced structure with minimal additional cost (Appendix C).

The LLRB tree exhibits very efficient search performance for existing elements. The structure's balanced nature ensures that the average search path remains short, leading to rapid retrieval of values. The graph clearly demonstrates that as the tree size increases, the increase in search time is minimal, which is a direct consequence of the logarithmic height maintained by the tree. The search operations for keys that do not exist in the tree are even more efficient. Since the tree remains well balanced, the search algorithm quickly determines the absence of a key by following a short, well-defined path. The results show that the tree can rapidly conclude unsuccessful searches, which minimizes unnecessary comparisons and traversals.

2.3 Scapegoat

The Scapegoat tree is subjected to random data insertion, sorted strings insertion, almost sorted strings, and how well a search algorithm functions in it.Tree performs well for random data as the tree will distribute keys evenly making the tree more balanced, which causes the scapegoating function to be rarely used in the process. On the graph scapegoat had increased in insertion time based on the number of inserts with some fluctuations at lower numbers but gradually stabilizing with higher numbers of elements.

It performs worst for sorted strings since Its performance fluctuates a lot as it inserts and sorts out elements when displayed on the graph. This occurs since a sorted string data set causes the tree to become very unbalanced with each insertion which will cause the tree to undergo balancing operations much more frequently leading to extended delays.

For almost sorted strings the performance varies depending on how "almost sorted" the data sets, as the closer the data is to being properly sorted the worst the performance is and vice versa. As the graph results show that almost sorted strings can perform faster than sorted strings with smaller data sets, their performance becomes closer to that of sorted strings fluctuating to be better or worse performing. However, on average almost sorted strings performs better than sorted strings also based on the graph (Appendix C).

Scapegoat also has a high performance for search algorithms since the trees will always be balanced after each data insertion due to the balancing of trees right as a byproduct of each insertion. The graph for a search algorithm, shows a relatively steady logarithmic increase in time for searching but as a very minimal amount of fluctuation between plots.

3 Comparative Assessment

For insertion performance, the AVL tree consistently demonstrates superior efficiency across different scenarios, including random, sorted, and almost sorted data. It maintains the lowest average insertion times, which increase gradually with tree size, indicating effective balancing. The LLRB tree shows moderate performance, with insertion times higher than AVL but lower than Scapegoat, and a more noticeable increase as the tree size grows. In contrast, the Scapegoat tree exhibits the highest insertion times, particularly with sorted data, suggesting less efficient balancing compared to AVL and LLRB.

In terms of search performance, the AVL tree again leads with the lowest average search times, maintaining efficiency as the tree size increases. The LLRB tree performs moderately, with search times higher than AVL but still better than Scapegoat. The Scapegoat tree shows the highest search times, indicating less efficient search operations.

Overall, while all the trees maintain an O (log n), the AVL tree emerges as the most efficient choice for balanced operations, consistently outperforming the other trees in both insertion and search tasks. The LLRB tree offers a good balance between performance and complexity, making it suitable for scenarios where AVL's strict balancing is not required. Meanwhile, the Scapegoat tree, while useful in certain contexts, tends to lag in performance, especially with larger or sorted datasets, making it less ideal for performance-critical applications.

Each tree excels in different scenarios. The AVL tree is well-suited for applications requiring frequent searches and updates, such as database indexing and memory management systems, due to its consistently efficient balancing and fast search times. The LLRB tree is a good choice for scenarios where simpler implementation is preferred over strict performance, such as educational tools or applications where moderate performance is acceptable. It offers a balance between complexity and efficiency. The Scapegoat tree, with its simpler balancing mechanism, can be useful in environments where memory usage is a concern, or in applications where insertion order is unpredictable, and occasional rebalancing is acceptable, such as certain types of caches or data logging systems.

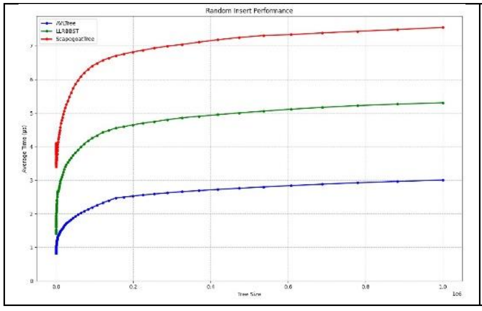
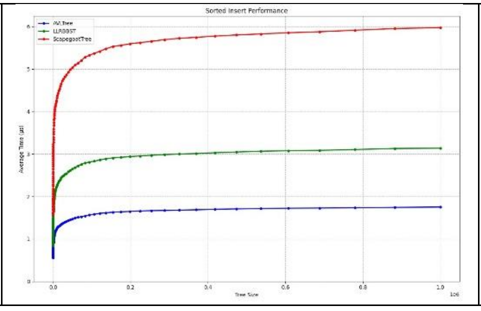
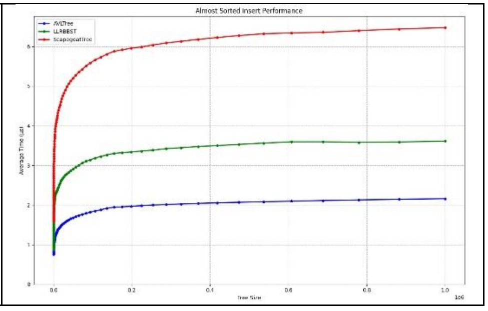
4 Team Contributions

Student Name	**Student Portico ID**	**Key Contributions**	**Share of work**[1]
Muhammad Asad Majeed	24174180	Implementation of LLRB Tree, plots of insertion by tree, report (2.2 and 3).	35%
Fang Ming Luan	24012383	Implementation of the AVL Tree, Design and implemented of TestDataGenerator, five	35 %
Brendan Loo	24170603	Scapegoat tree code, and report (2.3)	25 %
Hussain Mahmood		Attended the first meeting	5 %

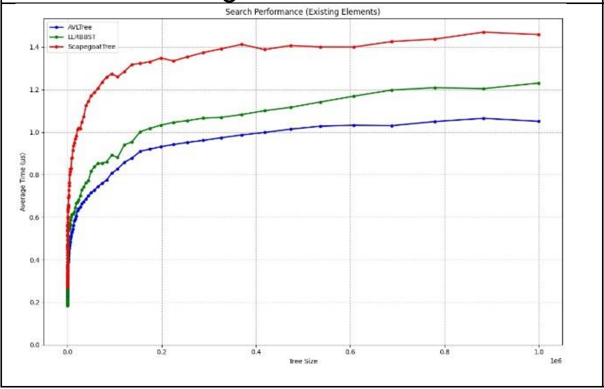
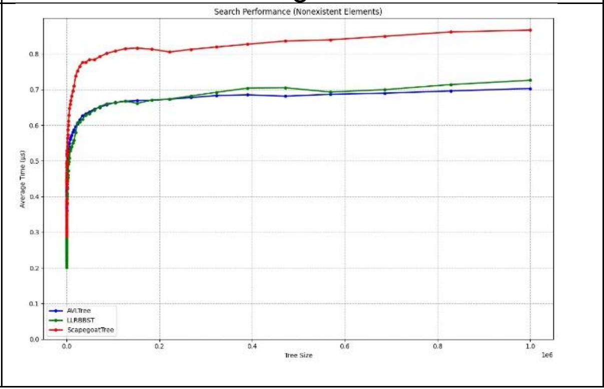
APPENDICES

Random Dataset	Sorted dataset	Almost Sorted Dataset

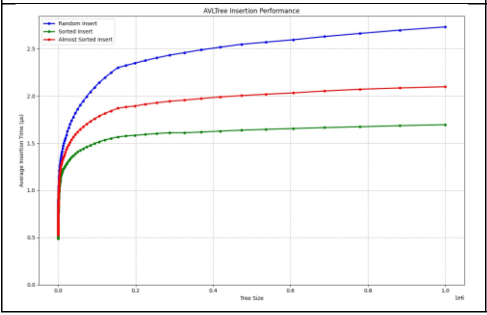
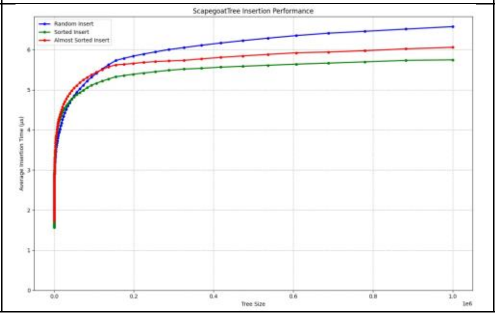
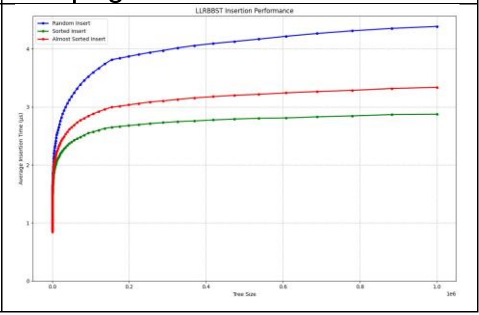
[1] This should be a **percentage**. For example, in a group of 4 students, if all members contributed equally (i.e., the ideal scenario), their share of work would be 25% each.

Appendix A: Insert Performances on different datasets.

Search Existing Elements	Search Non-Existing Elements
	

Appendix B: Search Performance

AVL Tree	LLRB	Scapegoat
		

Appendix C: Individual Insertion Performance by tree type