**COMP0005 Group Coursework**

**Academic Year 2024-25**

**Group Number: 10**

# Overview of Experimental Framework

## 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 diverse conditions. It consists of two primary components: TestDataGenerator for synthetic data generation and ExperimentalFramework for performance evaluation.

In this experiment, generated strings are all 5 characters and the dataset size to 1,000,000 strings. From a unique set of 1,000,000 random lowercase strings, four datasets are derived: (1) an unsorted dataset; (2) a sorted dataset (ascending); (3) a partially sorted dataset with 10% swaps; and (4) an uppercase and digit dataset for nonexistent element searches. The first three datasets, sharing identical elements but differing in sequence, are used to test insertion performance, while the fourth evaluates worst-case search performance.

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).

## Hardware/Software Setup for Experimentation

*(Section 1 should be about one page)*

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, Build 22631.2631), 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-mitigration techniques (e.g., repeated measurements, smoothing), ensured that performance differences reflected algorithmic efficiency, enabling a fair and accurate comparison.

# Performance Results

## 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.

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.

测评结果表明， AVL 树在应对不同序列的数据集（随机、排序、几乎排序）时，性能反应差异并不显著，插入时间均呈现平稳的对数增长趋势。 AVL 树通过局部旋转操作（复杂度 O(1)）高效应对有序输入的失衡，尽管旋转频率较高，但其优化效应在大规模数据下显著体现。理论上，AVL 树的插入涉及平衡因子检测与旋转调整，有序输入虽在树的同一侧（如右侧）连续添加节点导致失衡，但每次旋转的局部性与常数复杂度使得累积开销被均摊至 O(log n)。相比之下，随机输入的失衡位置分散，需更多高度检查或遍历，反增隐性开销。同时，有序输入增强了缓存局部性，因插入集中在某一区域（如右侧子树），节点连续访问提升缓存命中率，降低内存开销；而无序输入的跳跃式分布导致更高缓存未命中率与分支预测失败。因此，在大规模数据下，有序输入的局部性优势足以抵消旋转开销，使其时间与无序输入相当甚至更低。当数据量足够大时，对数高度的主导效应进一步缩小旋转的常数成本占比，结合旋转优化与缓存效应，有序输入性能得以持平甚至超越无序输入，充分展现 AVL 树的自适应能力与实现效率。

AVL 树在搜索已插入元素和不存在元素时均表现出高效、稳定且相近的性能，这源于其严格的高度平衡特性，保证了无论键值是否存在，搜索路径长度都受限于 O(log n) 的树高，且路径明确无冗余。成功搜索到达目标节点，失败搜索到达叶节点，二者深度相近，比较次数差异极小，体现了对称的高效性和可靠性。

## LLRBST

## Scapegoat

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.

It performs worst for sorted strings since it maximises the tree’s imbalance causing more frequent rebuilding of subtrees to rebalance the tree. For almost sorted strings it has a higher performance than with sorted strings but less than random data, as the tree is unbalanced not as often but still has to go through regular scapegoat rebalancing procedures. Scapegoat also has a high performance for searching since the trees always remain as balanced as possible due to the scapegoat mechanism keeping the tree height logarithmic.

*(Section 2 should be about two / two-and-a-half pages)*

# Comparative Assessment

*(Section 3 should be about one / one-and-a-half pages)*

# Team Contributions

|  |  |  |  |
| --- | --- | --- | --- |
| **Student Name** | **Student Portico ID** | **Key Contributions** | **Share of work[[1]](#footnote-0)** |
| Muhammad Asad Majeed |  |  | … % |
| Fang Ming Luan |  |  | … % |
| Brendan Loo |  |  | … % |
| Hussain Mahmood |  |  | … % |

*(Section 4 should be no more than half a page)*

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. [↑](#footnote-ref-0)