Advanced Data Structures - Introducing, Benchmarking, and Comparing AVL Trees, Red-Black Trees, and Treaps

HAMZA MUHAMMAD ANWAR, University of British Columbia Okanagan, Canada

This is the report for the second assignment of the COSC 520 (Advanced Algorithms) course. The objective of the assignment was to: (1) explore advanced data structures, and (2) benchmark algorithms. The three comparable advanced data structures chosen were AVL Trees, Red-Black Trees, and Treaps. Their functions and potential applications were detailed. Their time and space computational complexities were analyzed while performing various operations, and the results were plotted. The GitHub repository containing my implementations of this assignment can be found here: https://github.com/Hamza-Awr/COSC-520-Assignment-2.git.

Additional Key Words and Phrases: Algorithms, Advanced Data Structures, Binary Search Trees, AVL Trees, Red-Black Trees, Treaps, Time complexity, Space complexity

1 Introduction

In assignment 2 of COSC 520, we were tasked with choosing, analyzing, and comparing three advanced data structures in terms of their time and space computational complexities while performing the same function, and plotting the results. I chose AVL trees, Red-Black trees, and treaps. The algorithms and code necessary to generate them were all implemented using Python.

1.1 Motivation for choosing the data structures

Our COSC 520 project is related to a paper whose algorithm achieved a smaller time complexity for Dijkstra's algorithm in finding the shortest path in directed graphs. That algorithm maintains tree-like structures (shortest path tree) to store the nodes and weights of recent shortest paths found in different subproblems.

This usage of trees inspired me to look deeper into different tree data structures, so that when we have to implement that paper's shortest path algorithm, we'll have a better idea of which tree to use. Therefore, I chose two of the most popular trees: AVL trees, Red-Black Trees, and for the third, I wanted to try something unique, so I chose Treaps.

1.2 Parameters Used

In this report, the following parameters were used:

- $O(\cdot)$: Denotes the asymptotic upper-bound time complexity.
- n: Represents the number of elements in the dataset, or equivalently, the number of nodes in a tree.
- h: Denotes the height of the tree.

2 AVL Trees

Originally invented in 1962 by mathematicians Georgy Adelson-Velsky and Evgenii Landis (hence the name AVL), an AVL tree is a self-balancing binary search tree (BST) whose height difference of any subtree for any node cannot be more than one [1]. The height difference is referred to as the balance factor, and is calculated as follows:

$$\text{Balance Factor} = \text{left subtree height} - \text{right subtree height [5]}.$$

An AVL Tree balances itself in $O(1)$ using rotations of which there are four types [5]:

Author's Contact Information: Hamza Muhammad Anwar, hanwar02@student.ubc.ca, University of British Columbia Okanagan, Kelowna, British Columbia, Canada.

- Left-Left (LL) Rotation
- Right-Right (RR) Rotation
- Left-Right (LR) Rotation
- Right-Left (RL) Rotation

The rotations are named based on where a new node is added to the tree, which requires a rotation to rebalance the tree. For example, if a node is added to the right subtree of the left child, the rotation required to balance the tree after that insertion is called the Left–Right Rotation. Likewise, if a node is added to the left subtree of the right child, the rotation required to balance the tree after that insertion is called the Right–Left Rotation. An example of an AVL tree is shown in Figure 1.

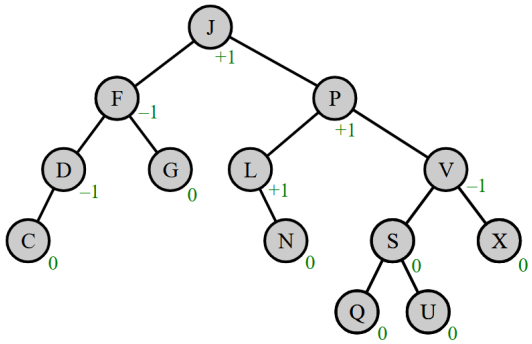


Fig. 1. An example of an AVL tree with balance factors [7]

2.1 Potential Applications of AVL Trees

Because AVL trees are able to provide quick lookups, insertions, and deletions, they are used where efficient data storage and retrieval are important, such as in file systems and power databases. They are also employed in memory management systems to optimize allocation and de–deallocation of resources.

AVL trees also see use in network routing tables to store and search routing entries efficiently in routers and switches. Search engines also employ them to maintain sorted indexes of terms for efficient keyword lookup.

2.2 Time Complexity

AVL trees perform four main operations: rotations, search, insert, and delete.

Rotations take $O(1)$ time, as they merely involve a few pointer changes without requiring any traversal through the tree, as they occur locally.

Searching is the same as searching through a binary search tree (BST), so it takes $O(h)$ time, where h is the height of the tree. Because it's a binary tree (each node has two children) $h = O(\log n)$, so the time complexity of search is $O(\log n)$ [4].

Insertion is also similar to a BST, where the new node is placed in a position where it is greater than the left node and smaller than the right node. After that, the balance factor of each node must be checked from the node to the root. If any node becomes unbalanced, a rotation on it is required to balance the tree again. Insertions also have a time complexity of $O(\log n)$, as you first search for the insertion spot, which takes $O(\log n)$ time, and perform a few rotations to rebalance the tree, which takes constant time.

Deletion, similar to insertion, takes $O(\log n)$ time. The node to delete is found ($O(\log n)$), deleted, and a few constant-time rotations are done to rebalance the tree.

2.3 Space Complexity

- Rotations take $O(1)$ space, as only a few pointers are rearranged and no recursion or extra memory is used.
- Search takes $O(1)$ space as it is iterative, not recursive. It only keeps a few variables and traverses down a path without storing it.
- Insertions have (auxiliary) space complexity of $O(\log n)$. This is for the recursion call stack, as the method of insertion used is recursive.
- Similarly, deletions also take $O(\log n)$ space due to using a recursive implementation.

3 Red-Black Trees

Red-black trees are another type of self-balancing binary search trees that ensure the tree height stays $O(\log n)$. They are less strict at balancing than AVL trees. The properties all Red-Black trees have are:

(1) Node Colour: Each node is either **red** or **black**.
(2) Root Property: The root is always black.
(3) Leaf Property: All leaves (NIL nodes) are black.
(4) Red Property: red nodes cannot have red children (no two reds in a row).
(5) Black-Height Property: For each node, all paths from the node to its descendant leaves contain the same number of black nodes. [1]

These rules ensure that the longest path of a red-black tree is no more than twice the shortest path, keeping operations efficient [1]. Figure 2 shows an example of a Red-Black tree.

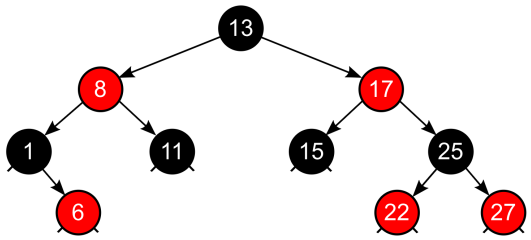


Fig. 2. An example of a Red-Black tree [8]

The types of rotations a red-black tree can do are:

(1) Left Rotation (Single Rotation): Pivot around a node x where the right child y becomes the parent of x. Used when the tree is right-heavy.
(2) Right Rotation (Single Rotation): Pivot around a node y where the left child x becomes the parent of y. Used when the tree is left-heavy.
(3) Left-Right Rotation (Double Rotation): First, left-rotate the left child, then right-rotate the node. Used to fix left-right imbalance.
(4) Right-Left Rotation (Double Rotation): First, right-rotate the right child, then left-rotate the node. Used to fix right-left imbalance.

The mechanics of rotations (left, right, left-right, right-left) are exactly the same in both AVL and Red-Black trees, as they both just pivot nodes around a child to maintain balance. However, in a

red-black tree, rotations are less common as they only need to occur when a rule has been violated. These rotations are combined with re-colours so that the properties of a red-black tree are restored.

3.1 Potential Applications of Red-Black Trees

Red-Black trees are often used to implement maps and sets, as these require frequent insertions and deletions [2]. Priority queues are another use case, where the sorting of elements based on priority is crucial. Red-Black trees are also used in event scheduling, where you need to keep events sorted by time to efficiently get the next event. File systems also employ their use to index file metadata (e.g., directories) for quick access. In graphics and game development, Red-Black Trees can be used for tasks like collision detection and pathfinding [2].

3.2 Time Complexity

The time complexities of Red-Black tree operations are similar to AVL trees.

Rotations (left/right) take $O(1)$ time.

Search takes $O(\log n)$ time as it is an iterative method. It is the same as searching through a BST, which takes $O(h)$ time where $h = O(\log n)$ and h is the height of the BST.

Insertion, also implemented iteratively, takes $O(\log n)$ time. Some rotations and recolouring may happen after insertions, but that takes constant time, so it has no effect overall.

Deletion, similar to insertion, takes $O(\log n)$ time. Some rotations and recolouring may happen after insertions, but that takes constant time, so it has no effect overall.

3.3 Space Complexity

The space complexities of Red-Black tree operations are also similar to AVL trees, so there is not much need to go into detail about them.

- Rotations (left/right) take $O(1)$ space. Single or double rotations are constant-time operations and don't need extra memory.
- Search (iterative) takes $O(1)$ space. The iterative method only uses constant extra memory.
- Insertion takes $O(1)$ space. Iterative insertion only needs a few pointers.
- Deletion also takes $O(1)$ space.

4 Treaps

Similar to AVL and Red-Black trees, treaps are also binary search trees, but they do not necessarily have a height of $O(\log n)$ [3]. They are randomized and are a hybrid between BSTs and heaps, hence their name. Heaps have a priority property, where a parent has higher priority than its children. Each treap node stores a key and a priority. The key is for BST ordering, while the priority is for heap ordering. There can be no duplicate keys. Unique random priorities are generated that help keep the tree balanced probabilistically, giving expected O(log n) operations. Figure 3 shows an example of a Treap.

Rotations similar to AVL or Red-Black tree rotations are used to restore the heap property based on priorities after insertion or deletion. The rotations used are:

- Right Rotation: If a left child's priority is higher than the parent, rotate right.
- Left Rotation: If a right child's priority is higher than the parent, rotate left.

In Treaps, rotations only restore heap property based on priorities; the BST property is never violated, unlike AVL or Red-Black trees.

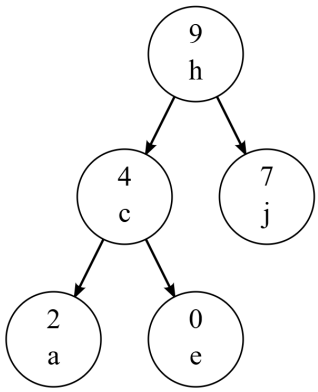


Fig. 3. An example of a Treap with alphabetic key and numeric max heap order [9]

4.1 Potential Applications of Treaps

Treaps can be used to implement simple, efficient ordered dictionaries, where they can store key-value pairs with fast *expected* O(log n) insertion, deletion, and search. In dynamic sets, treaps can be used to support operations like finding the k-th smallest element or rank queries efficiently. Because of its hybrid BST+heap nature, they may also be useful for priority-based task scheduling. Their simplicity also allows for a lighter implementation than AVL/Red-Black trees for BSTs. Treaps can also be used for maintaining authorization certificates in public-key cryptosystems [6].

4.2 Time Complexity

The *expected* time complexities of Treap operations Search, Insert, and Delete are $O(\log n)$. They are "expected" because they depend on the random priorities assigned to nodes, which determine the tree structure. The height of a Treap is not strictly balanced like an AVL or Red-Black tree, but with high probability, the randomized priorities produce a tree of height $O(\log n)$. Theoretically, individual runs can produce a worse shape. For example, the worst-case height = $O(n)$, causing the worst-case time complexity to be $O(n)$, but this is highly unlikely. Thus:

- Similar to the previous two BSTs discussed, Rotations take constant time because they are merely pointer updates.
- Search takes $O(\log n)$ time (expected), as it searches through a BST of height $O(\log n)$.
- Insert takes $O(\log n)$ time: traversing through the BST of height $O(\log n)$ to find the correct position to insert. Some rotations may need to happen to maintain the heap priority, but that takes constant time ($O(1)$).
- Delete takes $O(\log n)$ time for a similar reason: traversing through the BST of height $O(\log n)$ to find the correct position to delete. Again, a node may need to be rotated to maintain the heap priority, but that takes constant time ($O(1)$).

4.3 Space Complexity

The space complexities of Treap operations Search, Insert, and Delete are also $O(\log n)$. This is due to their recursive nature, so keeping that recursive stack takes $O(\log n)$ space. For Insert and Delete, some rotations may occur, but Rotations' space complexity is constant $O(1)$, so it doesn't have an effect.

5 Complexity Summary

A summary of the time and space complexities of all the data structures is given in Table 1.

Data Structure	Operation	Time Complexity	Space Complexity
AVL Tree	Rotation (single/double)	$O(1)$	$O(1)$
	Search	$O(\log n)$	$O(1)$ (iterative)
	Insertion	$O(\log n)$	$O(\log n)$ (recursive stack)
	Deletion	$O(\log n)$	$O(\log n)$ (recursive stack)
Red-Black Tree	Rotation (left/right)	$O(1)$	$O(1)$
	Search	$O(\log n)$	$O(1)$ (iterative)
	Insertion	$O(\log n)$	$O(1)$ extra (iterative)
	Deletion	$O(\log n)$	$O(1)$ extra (iterative)
Treap	Rotation (left/right)	$O(1)$	$O(1)$
	Search	$O(\log n)$ expected	$O(\log n)$ recursive stack
	Insertion	$O(\log n)$ expected	$O(\log n)$ recursive stack
	Deletion	$O(\log n)$ expected	$O(\log n)$ recursive stack

Table 1. Time and space complexities of AVL Trees, Red-Black Trees, and Treaps for main operations.

6 Implementation

The code was implemented in Python. The datasets were synthesized each time by randomly generating strings of five random ASCII characters. The number of usernames searched was kept constant at 1000 for each algorithm and data structure. It was not scaled with the value of n, as that would give a false comparison between the algorithms due to their differing time complexities.

The maximum value of n used was 10 million, as was required in the assignment. Similar to Assignment 1: The Login Checker Problem, the task I chose for this assignment was to insert, search for, and delete usernames in a generated dataset of usernames. The number of insertions was n, the size of the dataset, while the number of lookups and deletions was 1000. The length of each username was 6. Comprehensive unit tests were conducted to ensure that the data structures' implementations are able to handle all cases.

7 Comparison Analysis

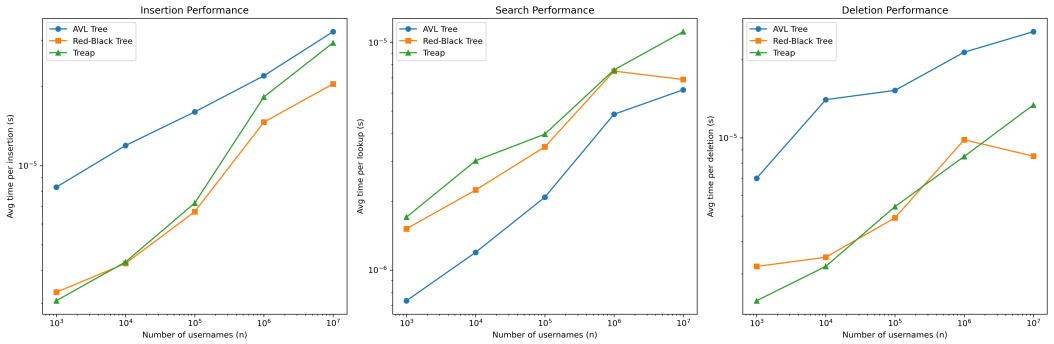


Fig. 4. A plot showing the comparison of time complexities of different operations in various data structures.

Figure 4 demonstrates the average operation times obtained from the implemented data structures for various operations.

From the overall shape of the plots, it can be seen that all operations of all three data structures took $O(\log n)$, which aligns with their theoretical time complexities.

The plotted results closely validate the theoretical expectations. In AVL Trees, Insertions and Deletions consistently take the longest time. This is primarily due to their recursive implementation, which introduces additional stack overhead. Search operations, on the other hand, are the fastest in AVL Trees. This is because AVL Trees maintain the strictest balance among the three structures, ensuring that the tree height never exceeds $O(\log n)$ and thus enabling rapid lookups.

Red-Black Trees, in contrast, demonstrate faster Insertions and Deletions compared to AVL Trees. This is largely because the implementation is iterative, avoiding the recursive stack overhead. Despite having the same theoretical $O(\log n)$ complexity as AVL Trees, this implementation detail gives Red-Black Trees a practical performance advantage for updates. However, their Search operation is slower than AVL Trees, due to their looser balancing criteria.

Treaps outperform AVL Trees in Insertions and Deletions due to their expected $O(\log n)$ height and efficient rotations guided by random priorities. However, they are the most loosely balanced structure, which negatively impacts Search performance. Lookups are slower in Treaps because maintaining the heap property during Insertions and Deletions can cause additional rotations and structural changes that are not directly optimized for key-based searches.

Overall, the results highlight the trade-offs between strict balancing, implementation choices, and probabilistic balancing: AVL Trees optimize for fast Search at the cost of slightly slower updates, Red-Black Trees strike a balance with efficient iterative updates, and Treaps offer expected-time efficiency with slightly compromised Search performance due to their randomized structure.

8 GitHub Repository

The GitHub repository containing my implementations of this assignment can be found here: https://github.com/Hamza-Awr/COSC-520-Assignment-2.git. The synthesized datasets are in the code for each algorithm itself. All instructions on the setup and running of the code can be found in the readme.md file.

Acknowledgments

Most of the code was written by the LLMs ChatGPT and Claude, and it was reviewed and executed by the author to ensure the correct results are obtained. Some parts of the report were also refined using ChatGPT.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[2] GeeksforGeeks. 09:20:38+00:00. Introduction to Red-Black Tree. https://www.geeksforgeeks.org/dsa/introduction-to-red-black-tree/.

[3] GeeksforGeeks. 11:57:47+00:00. Treap (A Randomized Binary Search Tree). https://www.geeksforgeeks.org/dsa/treap-a-randomized-binary-search-tree/.

[4] GeeksforGeeks. 12:47:31+00:00. Searching in Binary Search Tree (BST). https://www.geeksforgeeks.org/dsa/binary-search-tree-set-1-search-and-insertion/.

[5] GeeksforGeeks. 16:14:00+00:00. AVL Tree Data Structure. https://www.geeksforgeeks.org/dsa/introduction-to-avl-tree/.

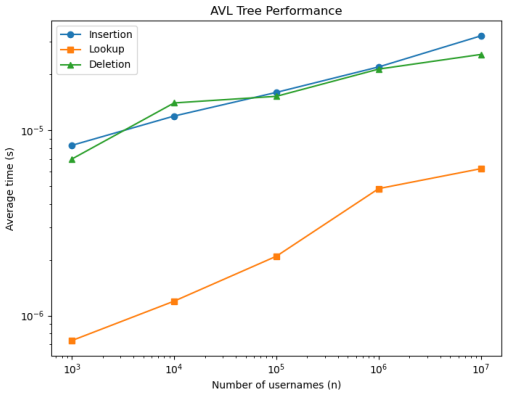
[6] M. Naor and K. Nissim. 2000. Certificate Revocation and Certificate Update. *IEEE Journal on Selected Areas in Communications* 18, 4 (April 2000), 561–570. doi:10.1109/49.839932

[7] Nomen4Omen. 2016. English: AVL Tree with Balance Factors.

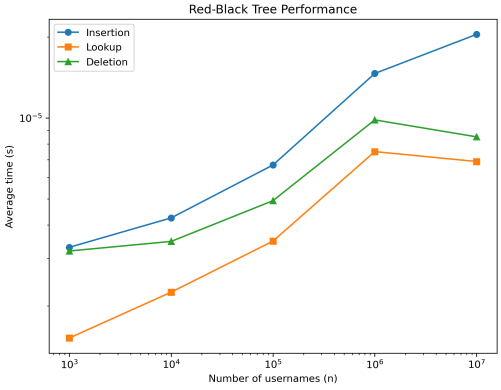
[8] Nomen4Omen. 2019. English: Example Red-Black Tree with Sockets.

[9] Qef. 2009. English: Treap with Alphabetic Key and Numeric Max Heap Order.

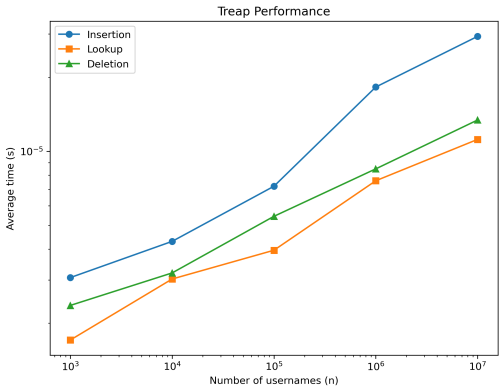
A Plots of the Time Complexities of Individual Data Structures



(a) The time complexity of Insertion, Search, and Deletion in the AVL tree implementation.



(b) The time complexity of Insertion, Search, and Deletion in the Red-Black tree implementation.



(c) The time complexity of Insertion, Search, and Deletion in the Treap implementation.

Fig. 5. Comparison of time complexities of data structures operations