Efficiency and Complexity Comparison of Red-Black Trees and AVL Trees in Data Structure

Juan Oloando Simanungkalit - 13524032
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
juanoloando.s@gmail.com , 13524032@std.stei.itb.ac.id

***Abstract*— This paper addresses the challenge of efficiently handling dynamic data operations, such as searching, deleting, and insertion, within tree-based data structures, specifically focusing on self-balancing binary search trees. The problem lies in understanding the trade-offs between different self-balancing algorithms to select the most suitable one for various workloads. The method employed involves a theoretical analysis and practical evaluation of two prominent self-balancing binary search tree algorithms: Red-Black Trees and AVL Trees. The paper compares their structural properties, balancing strategies, and operational efficiencies, particularly focusing on insertion, deletion, and search operations. The purpose of this paper is to provide a comprehensive understanding of the efficiency and complexity trade-offs between Red-Black Trees and AVL Trees.**

***Keywords*—AVL Tree, Red-Black Tree, Binary Seacrh Tree.**

I. INTRODUCTION

In the world of data structures, trees play a crucial role due to their accessibility in efficiently and hierarchically handling data. While other data structures like arrays, linked list, or hash tables offer various functionalities, trees hold a highly significant position in data structures due to their proficiency in hierarchical and efficient data management. Getting into the details, self-balancing binary search trees stand out by addressing efficiency concerns in dynamic operations like searching, deleting, and insertion, solidifying trees role as a primary foundation in numerous modern computer applications.

There are various types of algorithms that operate on binary search trees, with Red-Black Trees and AVL Trees being two of the most well-known. While these two algorithms share many similarities, they also have significant differences in how they maintain balance and handle operations. AVL Trees maintain a stricter balance by ensuring that the height difference between the left and right subtrees does not exceed one. This tight control often leads to quicker search times, but may require more frequent rebalancing. On the other hand, Red-Black Trees apply specific coloring rules to maintain a looser balance, which typically results in faster insertions and deletions due to fewer rotations.

This paper examines the trade-offs in efficiency and complexity between Red-Black Trees and AVL Trees through both theoretical analysis and practical evaluation. By comparing their structural properties and balancing strategies, the paper aims to identify which tree structure is better suited for various workloads and applications scenarios. A thorough understanding of these differences is essential for developers, students who want to study these two algorithms for their projects, and computer scientists in selecting the most appropriate data structure to achieve optimal performance.

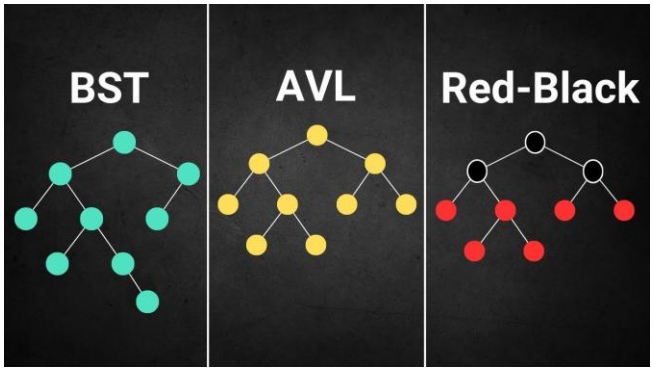

Figure 1.1 Binary Search Tree, AVL Tree, Red-Black Tree

(DeepDiveIntoBinary)

II. THEORETICAL BASIS

A. Definition and Components of Tree

A tree can be understood as an undirected graph that contains two crucial properties: it is entirely connected, and it contains no circuits. Therefore, the defining characteristics of a tree are its undirected nature, its connectivity, and the absence of any cycles within in structures. Rooted tree is a tree which a single node is chosen as the root, and its edges are given direction, transforming it into a directed graph.

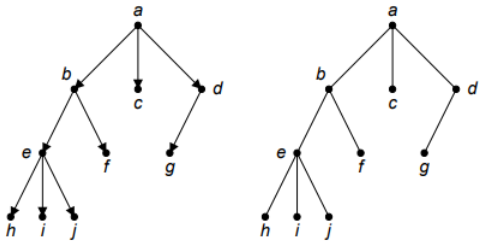

Figure 2.1 Rooted Tree

(https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf)

There are several terms or terminologies regarding trees that one must understand to comprehend Binary Search Trees. These terminologies are:

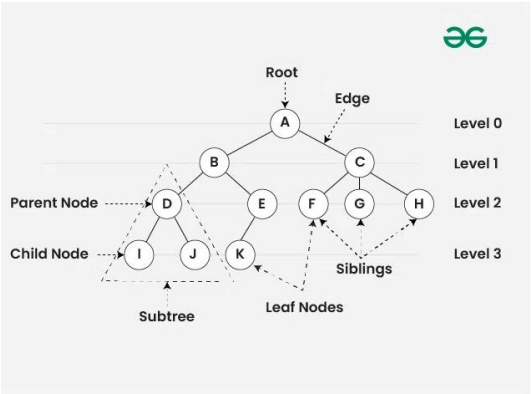

Figure 2.2 Basic Terminologies In Tree

(https://www.geeksforgeeks.org/introduction-to-tree-data-structure/)

1. Parent Node : The nod which is an immediate predecessor of a node is called the parent of that node. Example: B is the parent node of D and E
2. Child Node : The node which is the immediate successor of a node is called the child node of that node. Example: D and E are the child nodes of B.
3. Root Node : The topmost node of a tree or the noed which does not have any parent node is called the root node. Referring to the figure above, A is the root node.
4. Leaf Node : The nodes which do not have any child nodes. Example: I,J,K,F,G, and H are the leaf nodes of the tree.
5. Sibling : Children of the same parent node. Example: D and E are called siblings.
6. Ancestor of a node: Any prodecessor nodes on the path of the root to that node. Example: A and B are the ancestor nodes of the node E.
7. Descendant : A node x is a descendant of another node y if and only if y is an ancestor of x.
8. Level of a node : The count of edges on the path from the root node to that node. The root node, {A}, has level 0.
9. Internal Node : A node with at least one child.
10. Subtree : Any node of the tree along with its descendant.

B. Binary Search Tree

A Binary Search Tree (BST) is a fundamental data structure in computer science used to organize and store data in a sorted manner. Each node in a BST has at most two children: a left and a right child. The left child's value is always less than its parent node's value, while the right child's value is greater than or equal to its parent's value. This structure allows for more efficient search, insertion, and deletion operations on the data stored within the tree.

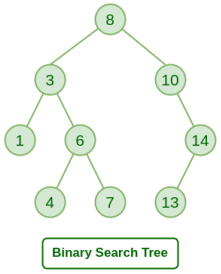

Figure 2.3 Binary Search Tree

(https://www.geeksforgeeks.org/binary-search-tree-data-structure/)

C. AVL Tree

An AVL Tree is a prime example of a self-balancing binary search tree. This innovative method was developed by computer scientist Georgi Maximovich Adelson-Velsky and Yevgeny Mikhailovich Landis in 1962.

An AVL tree is a type of binary search tree that maintains balance by ensuring that for every node, the height difference between its left and right subtrees is never more than one. If a insertion or deletion operation causes this balance to be disturbed, the tree automatically rebalances itself using AVL rotations. This process guarantees efficient performance for all tree operations.

The height of a subtree demonstrates how far the root is from the lowest node. Therefore, a subtree that contains only a root node has height of 0. A node's balance factor (BF) is calculated by subtracting the height of its left subtree from the height of its right subtree. For any non-existent subtrees, their height is considered to be -1 (which is one less than a subtree with just a single node).

$$BF(node) = H(node.right) - H(node.left)$$

There are three cases:

1. If the balance factor is < 0, the node is classified as *left-heavy.*
2. If the balance factor is > 0, the node is classified as *right-heavy*
3. A balance factor of 0 indicates a balanced node.

In an AVL tree, the balance factor at each node must have a value between -1, 0, or 1.

AVL rotations become necessary when an insertion or deletion operation causes the tree become unbalanced. There

are two types of rotations: right rotations and left rotations. The image below illustrates a right rotation and left rotation. The tree displayed the following nodes:

1. N: the root node where an imbalance was found
2. L: the child node of N
3. LL: the left child node of L
4. LR: the right child node of L
5. R: the right child node of N

From the tree below, there is a key piece of information:

LL(1) < L(2) < LR(3) < N(4) < R(5)

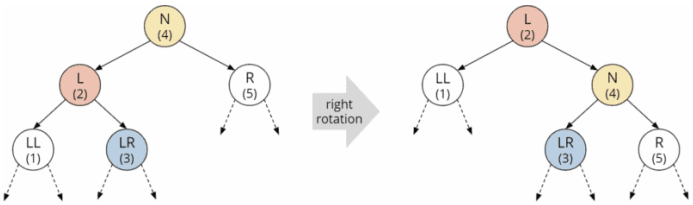

Figure 2.4 Right Rotation

(https://www.happycoders.eu/algorithms/avl-tree-java/)

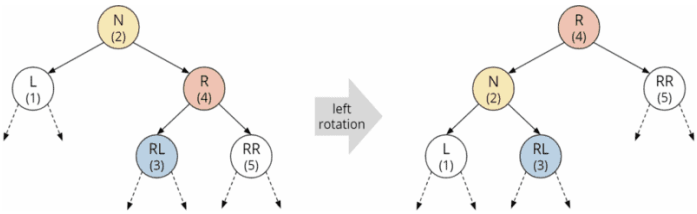

Figure 2.5 Left Rotation

(https://www.happycoders.eu/algorithms/avl-tree-java/)

To keep an AVL Tree balanced, whenever a node is inserted or deleted, the height and balance factor for all affected nodes must be recalculated, moving upward from the modification towards the tree's root. If this process reveals a node where the AVL invariant is broken (meaning its balance factor falls outside the acceptable range of -1,0, or 1) then the tree must be rebalanced. This rebalancing process falls into one of four categories:

1. Balancing a left-heavy node:
    a. Right rotation
    b. Left-right rotation
2. Balancing a right-heavy node:
    a. Left rotation
    b. Right-left rotation

D. Red-Black Tree

A red-black tree is a type of self-balancing binary search tree designed to automatically keep itself balanced. Each node within this tree is assigned either a red or black color. A specific set or rules governs how these colors are arranged; for instance, a red node cannot have red children. This color scheme helps the tree maintain its balance.

After nodes are inserted or deleted, complex algorithms are used to verify that these rules are still being followed. If any rules are violated, the tree is rebalanced by recoloring nodes and performing rotations to restore the required properties. These trees are often represented with NIL nodes, which are empty leaf nodes without values. These NIL nodes are crucial for the algorithms, particularly when determining the colors of related nodes like uncles or siblings.

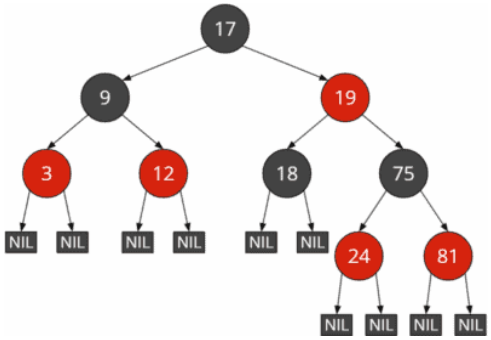

Figure 2.6 Red-Black Tree

(https://www.happycoders.eu/algorithms/red-black-tree-java/)

A red-black tree maintains balance through a set of strict rules governing node colors. Each node is either red or black, and all NIL leaves are strictly black. A critical rule requires that a red node may not have red children, and furthermore, every path from given node to its descendant leaves must contain an identical count of black nodes. While the root is typically black, the particular rule is often omitted in literature because its observance is often enforced by other rules. The specific rule's implementation typically results in only a minor code difference in operations.

The height of a red-black tree is defined as the maximum number of nodes from the root to a NIL leaf, excluding the root itself. A significant property derived from the rules is that the longest path from the root to any leaf is at most twice the length of the shortest path.

Inserting and deleting nodes in a red-black tree largely follows the standard procedures for binary search trees. However, after these operations, the tree's observance to its red-black rules must be checked. if any rules are violated, the tree is rebalanced by recoloring nodes and performing rotations. These rotations function identically to those used in AVL trees.

III. COMPARISON OF AVL TREE AND RED BLACK TREE

Both AVL tree and red-black tree are self-balancing binary search trees that guarantee a worst-case height of $O(\log n)$ for n nodes, which means operations like search, insertion, and deletion will take at most $O(\log n)$ time. While AVL tree maintains a stricter balance, often leading to faster search times, red-black tree employs a looser balancing strategy using color rules. This difference in balancing strictness is key to

understanding why the more complex red-black tree is still relevant, despite the simpler implementation of AVL tree. The comparison will focus on two primary primitive operations: insertion and deletion. The tool to used to visualize these two algorithms is Python.

A. Algorithm for AVL Tree

```python
# Inisialisasi Tree Node
class TreeNode(object):
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1
```

Figure 3.1 Create a Tree Node (AVL Tree)

(https://www.programiz.com/dsa/avl-tree)

The TreeNode class serves as the foundational building block for constructing an AVL tree. Each TreeNode object contains four essential attributes: a key to store the node's value, left and right pointers to reference its child nodes (initialized as None), and a height value (initialized to 1) that tracks the depth of the subtree rooted at that node. The key determines the node's position within the tree, adhering to the binary search tree property where all keys in the left subtree are smaller than the node's key, and all keys in the right subtree are larger. The height attribute is critical for maintaining the AVL tree's balance, as it enables the calculation of the balance factor—the difference between the heights of the left and right subtrees. By ensuring this balance factor remains within the range of -1,0,or 1, the AVL tree guarantees efficient operations with logarithmic time complexity. When a new node is created, it starts as a leaf node with no children, and its height is set to 1.

```python
class AVLTree(object):

    # Function to insert a node
    def insert_node(self, root, key):

        # Find the correct location and insert the node
        if not root:
            return TreeNode(key)
        elif key < root.key:
            root.left = self.insert_node(root.left, key)
        else:
            root.right = self.insert_node(root.right, key)

        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))

        # Update the balance factor and balance the tree
        balanceFactor = self.getBalance(root)
        if balanceFactor > 1:
            if key < root.left.key:
                return self.rightRotate(root)
            else:
                root.left = self.leftRotate(root.left)
                return self.rightRotate(root)

        if balanceFactor < -1:
            if key > root.right.key:
                return self.leftRotate(root)
            else:
                root.right = self.rightRotate(root.right)
                return self.leftRotate(root)

        return root
```

Figure 3.2 Algorithm for Insertion (AVL Tree)

(https://www.programiz.com/dsa/avl-tree)

The AVL tree insertion algorithm is a carefully designed process aimed at maintaining the tree's balance as new nodes are introduced. It begins with the standard binary search tree (BST) insertion method, where the algorithm recursively searches for the new node's appropriate position. If the designated spot is empty, a new node is created. Conversely, if the spot is occupied, the new key is compared with the current node's key, and the insertion proceeds recursively into either the left or right subtree to uphold the BST property.

Once the new node is placed, the algorithm updates the height of the current node by adding one to the maximum height of its child subtrees. This height management is vital for the subsequent balance checks. The balance factor is then computed by subtracting the height of the right subtree from that of the left subtree. Should this balance factor fall outside the acceptable range of -1 to 1, the tree undergoes rebalancing through specific rotations. There are four distinct rotation scenarios, each addressing a particular type of imbalance: a right rotation for a left-left imbalance, a left-right rotation for a left-right imbalance, a left rotation for a right-right imbalance, and a right-left rotation for a right-left imbalance. These rotations effectively reorganize the tree's structure while preserving its BST properties, thereby ensuring that the height difference between any node's subtrees never exceeds one. The algorithm concludes by returning the modified subtree, allowing these balancing adjustments to propagate up the recursive call stack and maintain overall tree stability. This combination of recursive insertion, height tracking, and conditional rotations guarantees that AVL tree operations consistently achieve $O(\log n)$ time complexity.

```python
class AVLTree(object):
    def delete_node(self, root, key):

        # Find the node to be deleted and remove it
        if not root:
            return root
        elif key < root.key:
            root.left = self.delete_node(root.left, key)
        elif key > root.key:
            root.right = self.delete_node(root.right, key)
        else:
            if root.left is None:
                temp = root.right
                root = None
                return temp
            elif root.right is None:
                temp = root.left
                root = None
                return temp
            temp = self.getMinValueNode(root.right)
            root.key = temp.key
            root.right = self.delete_node(root.right,
                                          temp.key)
        if root is None:
            return root
```

Figure 3.3 Algorithm for Deletion (AVL Tree)

(https://www.programiz.com/dsa/avl-tree)

Deletion within an AVL tree is managed by a structured algorithm that strictly upholds the tree's balanced configuration during node removal. The process initiates by adhering to standard binary search tree (BST) deletion principles, recursively seeking the node targeted for removal. If the node is not present (i.e., the root is null), the operation simply terminates. Upon locating the desired node, its removal is handled based on its child count: nodes with no left child are replaced by their right child; those with no right child are replaced by their left child; and nodes possessing both children necessitate finding their in-order successor (the smallest value in the right subtree), transferring its value to the current node, and then recursively deleting that successor from the right subtree. This strategy ensures the BST property remains preserved.

Subsequent to the node's removal, the algorithm undertakes crucial rebalancing adjustments. It first updates the height of the affected node, computing it as one plus the maximum height of its remaining child subtrees. The balance factor is then derived by comparing the heights of the left and right subtrees. Should an imbalance manifest—that is, if the balance factor exceeds 1 or falls below -1—the algorithm triggers the appropriate rotation(s) to restore equilibrium. Four distinct rotation patterns address specific imbalances: a simple right rotation for left-left scenarios, a left-right double rotation for left-right cases, a simple left rotation for right-right situations, and a right-left double rotation for right-left imbalances. These rotations are meticulously designed to maintain both the BST property and the AVL balance condition while minimizing structural alteration. The algorithm concludes by returning the potentially restructured subtree, allowing these changes to propagate upward through the recursive call stack to ensure balance across the entire tree. This integrated approach—combining recursive node removal with height management and conditional rotations—guarantees that AVL tree operations consistently achieve optimal $O(\log n)$ time complexity and retain their balanced characteristics.

For example, the initial AVL tree contains the following nodes: [50,12,52,10,25,61,8,11]. The objective is to insert the value [30] into this tree while maintaining the AVL property.

```
R----50
     L----12
     |    L----10
     |    |    L----8
     |    |    R----11
     |    R----25
     R----52
          R----61
After Insertion:
R----50
     L----12
     |    L----10
     |    |    L----8
     |    |    R----11
     |    R----25
     |         R----30
     R----52
          R----61
PS C:\ITB\informatika\Matdis\Codes\rb python>
```

Figure 3.4 Initial and Final AVL Tree Insertion

The insertion of the value 30 into the initial AVL tree [50, 12, 52, 10, 25, 61, 8, 11] follows a systematic process to maintain the tree's balanced structure. The algorithm begins by performing a standard BST insertion, traversing from the root (50) to the appropriate position. Since 30 is less than 50 but greater than 12 and 25, it is inserted as the right child of node 25. Following insertion, the heights of affected nodes (30, 25, 12, and 50) are recalculated to reflect the structural changes.

The critical rebalancing phase occurs when checking the balance factors after insertion. Node 12, with a balance factor of 2 (left subtree height of 3 minus right subtree height of 1), becomes unbalanced. This specific imbalance represents a Right-Right (RR) case, where the right subtree of node 12's right child (25) contains the newly inserted node (30). To correct this, a left rotation is performed on node 12. The rotation restructures the tree by making node 25 the new parent of node 12, with node 12 becoming the left child of node 25 and node 30 remaining as the right child. This rotation reduces the height difference between subtrees while preserving the BST property.

The rotation successfully restores balance to the tree, maintaining the AVL property where no node has a balance factor exceeding ±1. The final tree structure demonstrates how AVL trees dynamically adjust through rotations to ensure optimal performance, with all operations maintaining $O(\log n)$ time complexity. This example illustrates the essential self-balancing mechanism of AVL trees, where insertions are followed by height updates and necessary rotations to preserve the tree's balanced state.

```
R----50
     L----12
     |    L----10
     |    |    L----8
     |    |    R----11
     |    R----25
     |         R----30
     R----52
          R----61
After Deletion:
R----50
     L----25
     |    L----10
     |    |    L----8
     |    |    R----11
     |    R----30
     R----52
          R----61
PS C:\ITB\informatika\Matdis\Codes\rb python>
```

Figure 3.5 Initial and Final AVL Tree Deletion

The deletion of node 12 from the given AVL tree, which initially contains [50, 12, 52, 10, 25, 61, 8, 11, 30], follows a precise procedure to ensure the tree remains balanced. Since node 12 has both a left child (10) and a right child (25), it is replaced by its in-order successor, which is node 25. Subsequently, node 25 is removed from its original location. This causes a structural adjustment where node 25 takes node 12's former position, with node 10 becoming its left child, and nodes 11 and 30 relocating within the right subtree.

After this initial deletion, the algorithm proceeds to update the heights of affected nodes and check their balance factors. A potential imbalance often arises at node 50 (the tree's root) because its left subtree's height has decreased. The balance factor for node 50 is calculated by subtracting the height of its right subtree (rooted at 52) from the height of its left subtree (now rooted at 25). If this balance factor moves outside the acceptable range of ±1, rotations are then initiated to restore balance.

In this specific example, two main scenarios might necessitate rotation:

- Should node 50 become left-heavy (balance factor greater than 1), a right rotation would be performed. This action would promote node 25 to the new root, with node 50 becoming its right child.
- If, however, node 25's right subtree becomes too heavy (balance factor less than -1), a left-right double rotation might be required. This involves first a left rotation on node 25's right child (node 30), followed by a right rotation on node 25 itself.

These corrective rotations are crucial for maintaining the AVL property, ensuring that no node's balance factor deviates beyond ±1. This preservation of balance is key to upholding the tree's efficient $O(\log n)$ time complexity for all operations. The precise rotations employed are determined by the exact height differences observed after the deletion, systematically restoring balance while fully retaining the binary search tree property.

B. Algorithm for Red-Black Tree

```
# Inisialisasi Tree Node
class Node():
    def __init__(self, item):
        self.item = item
        self.parent = None
        self.left = None
        self.right = None
        self.color = 1
```

Figure 3.6 Create a Tree Node (Red Black Tree)
(https://www.programiz.com/dsa/red-black-tree)

The fundamental structure of a Red-Black tree's Node class is quite similar to that of an AVL tree, with the key distinctions being the inclusion of two additional attributes: parent and color. In an AVL tree, each node typically stores its key, pointers to its left and right children, and a height attribute. This height is crucial for maintaining the tree's balance by ensuring that the height difference between any left and right subtree does not exceed one.

On the other hand, the Red-Black Tree's Node class contains an item (or key), left and right child pointers, a parent pointer, and a color attribute (where '1' often denotes red and '0' signifies black). The parent pointer is vital for efficient movement within the tree and for rebalancing operations, as it allows access to a node's ancestor. The color attribute, meanwhile, is essential for enforcing the Red-Black Tree's specific balancing rules: primarily, that no two consecutive red nodes are allowed (meaning a red node cannot have a red child), and that every path from the root to any leaf must contain an identical count of black nodes.

These constraints work together to ensure that the Red-Black Tree remains approximately balanced. This balancing is less strict than that of an AVL tree, which typically leads to fewer structural adjustments (rotations) during insertions and deletions. Therefore, while AVL trees prioritize strict height balance for potentially faster search operations, Red-Black Trees are optimized for more efficient modifications (insertions and deletions) by relying on these color-based rules. The inclusion of the parent and color attributes in the Red-Black Tree's node structure is fundamental to its distinct self-balancing mechanism, setting it apart from the AVL tree's height-centric approach.

```
class RedBlackTree():
    def insert(self, key):
        node = Node(key)
        node.parent = None
        node.item = key
        node.left = self.TNULL
        node.right = self.TNULL
        node.color = 1

        y = None
        x = self.root

        while x != self.TNULL:
            y = x
            if node.item < x.item:
                x = x.left
            else:
                x = x.right

        node.parent = y
        if y == None:
            self.root = node
        elif node.item < y.item:
            y.left = node
        else:
            y.right = node
```

```
class RedBlackTree():
    def fix_insert(self, k):
        while k.parent.color == 1:
            if k.parent == k.parent.parent.right:
                u = k.parent.parent.left
                if u.color == 1:
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
                else:
                    if k == k.parent.left:
                        k = k.parent
                        self.right_rotate(k)
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    self.left_rotate(k.parent.parent)
            else:
                u = k.parent.parent.right

                if u.color == 1:
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
```

Figure 3.7 Algorithm for Insertion (Red-Black Tree)

(https://www.programiz.com/dsa/red-black-tree)

The insertion process in a Red-Black Tree consists of two fundamental phases: standard binary search tree insertion followed by tree rebalancing to maintain the tree's critical properties. The process begins by creating a new node containing the key value, which is initialized as red (color = 1) with its left and right children set to null (TNULL). The algorithm then traverses the tree from the root downward, comparing the new node's value with existing nodes to determine its proper position according to BST rules - moving left when the new value is smaller and right when it is larger.

Once the appropriate position is found, the new node is linked to its parent. Special cases are handled immediately: if the new node becomes the root, it is recolored black to satisfy

the root property, and if the parent is the root (meaning no grandparent exists), the insertion completes without further adjustments. For all other cases, the fix_insert method is invoked to address potential violations of Red-Black Tree properties, primarily the red-red conflict where a red node has a red parent.

The fix_insert method systematically resolves these violations through a combination of recoloring and rotations. Three main scenarios are considered: when the uncle node is red (Case 1), requiring simple recoloring of the parent, uncle, and grandparent; when the new node forms a triangle configuration with its parent and grandparent (Case 2), necessitating an initial rotation to convert it to Case 3; and when they form a straight line (Case 3), resolved with a single rotation and recoloring. These operations ensure the tree maintains all essential properties: the root remains black, no two red nodes are adjacent, and all paths from any node to its descendant leaves contain the same number of black nodes, thereby guaranteeing balanced performance with $O(\log n)$ time complexity for all operations. Similarly, after a node is deleted, the tree will attempt to rebalance itself.

Using the same example as the AVL tree, the value 30 will be inserted into the initial tree (50, 12, 52, 10, 25, 61, 8, 11).

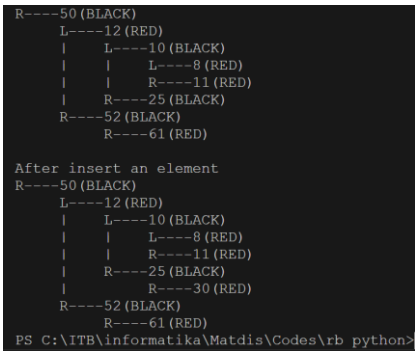

Figure 3.8 Initial and Final Red Black Tree Insertion

To integrate the value 30 into the Red-Black Tree (initially containing nodes [50, 12, 52, 10, 25, 61, 8, 11]), a structured procedure is followed to uphold the tree's crucial balancing characteristics. Node 30 is initially added as a red node, correctly positioned as the right child of 25 according to standard binary search tree rules. This immediate placement, however, promptly results in a red-red violation, as both node 30 and its parent 25 are red.

To address this conflict, the tree assesses the "uncle" of node 30—the sibling of its parent 25. This node, 10, is black. Given the black uncle, the tree initiates a sequence of rotations and recoloring operations. Initially, a left rotation is performed at node 25, which converts the right-heavy subtree into a linear arrangement. Subsequently, nodes are recolored: node 30 turns black, and node 12 changes to red. This is then followed by a right rotation at node 12, aiming to fully restore the tree's balance.

The resulting structure successfully upholds all Red-Black Tree properties: the root (50) remains black, no two red nodes are adjacent, and every path from the root to any leaf contains an equal count of black nodes. This entire procedure exemplifies the Red-Black Tree's efficient self-balancing capability through judicious rotations and color adjustments post-insertion. Such dynamic reorganization ensures optimal $O(\log n)$ time complexity for all operations and safeguards the tree's structural integrity, thereby averting the performance decline that an unbalanced binary search tree might experience.

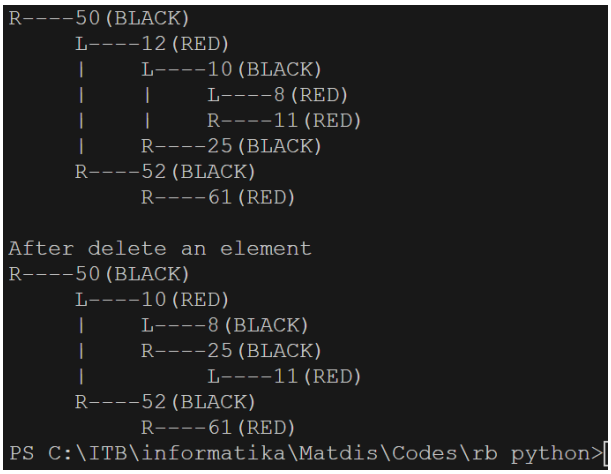

Figure 3.9 Initial and Final Red Black Tree Deletion

The removal of node 12 from the Red-Black Tree adheres to a carefully organized process designed to preserve the tree's essential characteristics. Initially, the algorithm identifies node 12, which possesses two children (node 10 and node 25). Given that node 12 is red and its children are black, the deletion can be simplified by substituting node 12 with its in-order successor, node 25. Following the removal of node 12, node 25 is promoted to assume its former position, while node 10 is retained as the left child.

This structural alteration, however, introduces a potential imbalance that necessitates correction through recoloring and, if required, rotations. The tree then executes a color flip: node 25 changes from black to red, and node 10 similarly changes from black to red, thereby maintaining the crucial black height property. If this recoloring were to result in a red-red violation between node 10 and its new parent, subsequent rotations would be performed for correction. In this particular instance, however, no additional rotations are necessary, as node 10's parent (node 25) remains black.

The resulting tree successfully maintains all Red-Black properties: the root (50) remains black, no red nodes are adjacent, and all paths from the root to the leaves contain an identical number of black nodes.

C. Comparison of Two Algorithms

Red-Black Trees and AVL Trees represent two distinct approaches to self-balancing binary search trees, each with unique advantages tailored to different computing scenarios. The fundamental difference lies in their balancing mechanisms - Red-Black Trees employ a color-coding system (red or black

nodes) with specific rules about node coloration and black node distribution, while AVL Trees maintain strict height balance through precise balance factors (-1, 0, or 1) for each node. This structural divergence leads to significant performance variations: AVL Trees, with their stricter balancing, guarantee a more optimal tree height of approximately 1.44 log(n) compared to Red-Black Trees' 2 log(n), making AVL Trees approximately 20-25% faster for lookup operations. However, this advantage comes at a cost - AVL Trees typically require more frequent and complex rotations during insertions and deletions, sometimes needing $O(\log n)$ rotations per operation, whereas Red-Black Trees usually require at most two rotations and often just simple recoloring. Storage requirements also differ substantially, with Red-Black Trees needing only a single bit per node for color information, while AVL Trees must store integer values (typically 2-4 bytes) for height or balance factor data. These characteristics lead to distinct application domains: Red-Black Trees dominate in systems requiring frequent modifications, such as language libraries (C++'s map/set, Java's TreeMap) and filesystem implementations, where their efficient insertion/deletion performance outweighs slightly slower searches. Conversely, AVL Trees excel in read-intensive environments like database indexing and real-time systems where maximum search speed is crucial and the data changes less frequently. Both maintain $O(\log n)$ time complexity for all operations, but their different balancing philosophies make each uniquely suited to specific performance requirements in computer science applications. The following table presents a summary of the distinctions between the two programs.

Comparison Indicators	AVL Tree	Red-Black Tree
Balance Factor	Each node has a balance factor whose value is between -1,0, or 1	It does not have a balance factor
Balancing	Take more processing for balancing	Take less processing for balancing. The maximum number of rotations is two.
Lookups	AVL Trees provide faster lookups than Red-Black Trees because they are more strictly balanced	Red-Black Tree has fewer lookups because they are not strictly balanced.
Color	There is no color of the node	The color of the node is either Red or Black
Insertion and Deletion	AVL Trees provide complex insertion and deletion operations as more rotations are done due to relatively strict balancing	Red-Black Trees provide faster insertion and deletion operations than AVL Trees as fewer rotations are done due to relaxed balancing
Storage	AVL Trees store balance factors or heights with each node. Therefore, requiring storage for an integer per node	Red-Black Tree requires only 1 bit of information per node
Searching	AVL Trees provide efficient searching	Red-Black Trees does not provide efficient searching
Applications	For indexing large records in databases, for searching in large databases, etc.	To implement finite maps, to implement Java packages, to implement Standard Template Libraries (STL) in C++: multiset,map,multimap, etc.

APPENDIX

Source code: https://github.com/kalitz23/Makalah-Matdis.

ACKNOWLEDGMENT

I extend my deepest gratitude to Almighty God for the blessings and guidance that allowed me to complete this paper. I also wish to thank my friends and family for their unwavering support during the preparation of this paper. Special thanks are due to Dr. Rinaldi and Mr. Arrival Dwi Sentosa, M.T., our Discrete Mathematics lecturers, whose profound knowledge and insights were instrumental in this work.

REFERENCES

[1] Code Visualizer, "Red-Black Tree," [Online]. Available: https://www.programiz.com/dsa/red-black-tree. [Accessed 20 June 2025].

[2] Code Visualizer, "AVL Tree," [Online]. Available: https://www.programiz.com/dsa/avl-tree. [Accessed 19 June 2025].

[3] GeeksforGeeks, "Red Black Tree vs AVL Tree," 10 November 2022. [Online]. Available: https://www.geeksforgeeks.org/dsa/red-black-tree-vs-avl-tree/. [Accessed 18 June 2025].

[4] HappyCoders, "Algorithm Red-Black Tree," [Online]. Available: https://www.happycoders.eu/algorithms/red-black-tree-java/. [Accessed 19 June 2025].

[5] HappyCoders, "AVL Tree Algorithm," [Online]. Available: https://www.happycoders.eu/algorithms/avl-tree-java/#AVL_Tree_Implementation_in_Java. [Accessed 19 June 2025].

[6] GeeksforGeeks, "Binary Search Tree," 19 June 2025. [Online]. Available: https://www.geeksforgeeks.org/binary-search-tree-data-structure/. [Accessed 20 June 2025].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Jakarta, 20 Juni 2025

Juan Oloando 13524032