A Comparative Analysis of Self-Balancing Binary Search Trees

Special Mathematics Lecture: Graph Theory (Spring 2024)

Yijiang Liu

July 25, 2024

A Binary Search Tree (BST) is an efficient data structure for data storage and quick retrieval. To maintain its efficiency, a BST needs to stay balanced, meaning the height difference between its left and right subtrees should not be too large. Otherwise, a BST may degrade into a linked list, causing the time complexity of search, insert, and delete operations to worsen from O(log n) to O(n). This report introduces several typical methods of balanced BST.

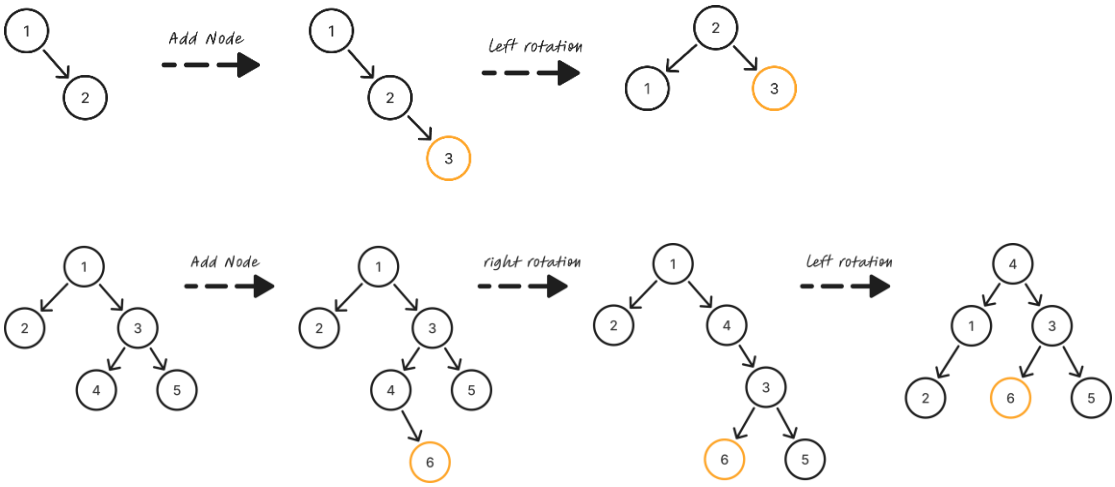
1. AVL Tree

The AVL tree is the earliest invented self-balancing binary search tree, named after G.M. Adelson-Velsky and E.M. Landis, who first proposed the concept of balanced binary trees.

The AVL tree is based on the concept of the balance factor (bf), which is defined as the depth of the left subtree minus the depth of the right subtree at a given node. An AVL tree maintains a balance factor at each node, ensuring the absolute value of the height difference between the left and right subtrees does not exceed 1 ($-1 \leq bf \leq 1$).

After inserting or deleting a node, necessary rotations are performed to maintain balance.

For example, after an insertion that causes the AVL tree to lose balance, there are four possible states: the insertion is in the left subtree (L) of the left child (L) of the grandparent node T, the right subtree (R) of the left child (L) of T, and their symmetric cases. For LL or RR imbalances, a single right or left rotation (also called zig or zag) on the grandparent or parent node usually restores balance locally, and thus globally. For LR or RL imbalances, a right (left) rotation on the parent followed by a left (right) rotation on the grandparent is required, as illustrated in the diagram.



2. Red-Black Tree

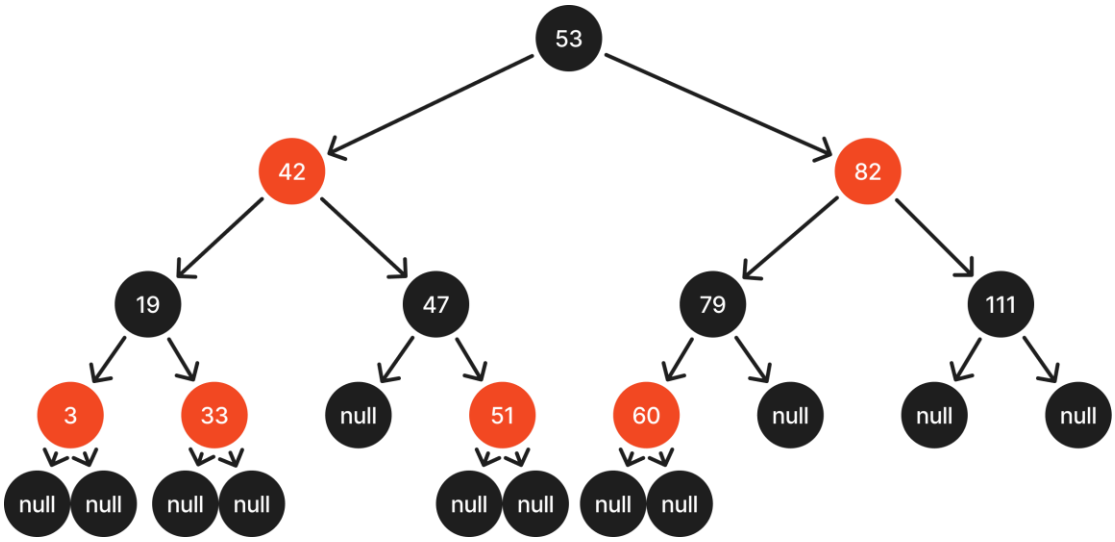
The Red-Black Tree is another self-balancing binary search tree, proposed by Rudolf Bayer in 1972. Each node in a Red-Black Tree has an additional color attribute (red or black) and maintains balance through a series of rules (e.g., the root must be black, red nodes must have black children) and rotations.

2.1 Introduction to Red-Black Trees

A Red-Black Tree is a self-balancing binary search tree and is an efficient search tree. Initially proposed by Rudolf Bayer in 1972, it was originally known as symmetric binary B-trees. Later, in 1978, Leo J. Guibas and Robert Sedgewick modified it to the current Red-Black Tree structure. A Red-Black Tree can perform search, insert, and delete operations in O(log n) time.

Why do we need Red-Black Trees when we already have AVL Trees, which are equally efficient? Consider a randomly inserted binary search tree, which tends to balance itself, making operations (search, insert, delete) efficient with a time complexity of O(log n). However, if the data inserted is ordered (increasing or decreasing), all nodes will be on the right or left side of the root, turning the BST into a linked list with a time complexity of O(n). Thus, the BST's time complexity can range from O(log n) to O(n) depending on the data.

In contrast, a Red-Black Tree maintains approximate balance without the strict balance factor concept of AVL Trees. It uses five properties to maintain a balanced structure, enhancing overall performance without adhering to a strict balance factor. This approach, by forgoing local node balance, achieves higher global efficiency.



2.2 Red-Black Tree Implementation

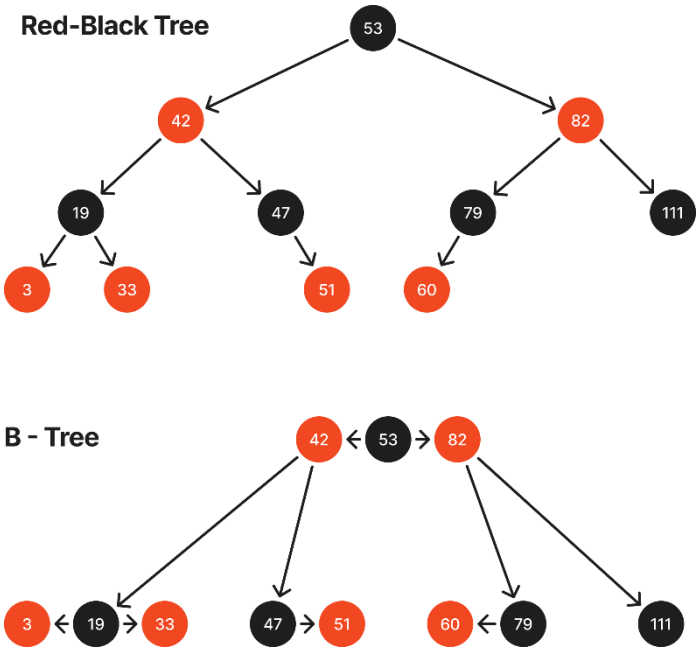
The structure of a Red-Black Tree is based on a binary search tree, with an additional bit at each node to record its color, either RED or BLACK. By constraining the colors along any path from the root to the leaves, Red-Black Trees ensure that the longest path is no more than twice the shortest path. This ensures approximate balance (the shortest path consists entirely of black nodes, while the longest path alternates red and black nodes, doubling the shortest path length).

A Red-Black Tree satisfies the following properties:

1. Nodes are either red or black.

2. The root is black.
3. All leaves (external nodes, or null nodes) are black. These leaf nodes are the null nodes at the bottom of the tree.
4. Red nodes must have black children, and their parents must be black, meaning no two consecutive red nodes are allowed along any path from the root to the leaves.
5. Every path from any node to its descendant leaves contains the same number of black nodes.

(By the way, combining all the red nodes into their black parent nodes in a Red-Black Tree results in an equivalent data structure: a B-tree, commonly used in computer disk indexing structures to reduce I/O operations.)



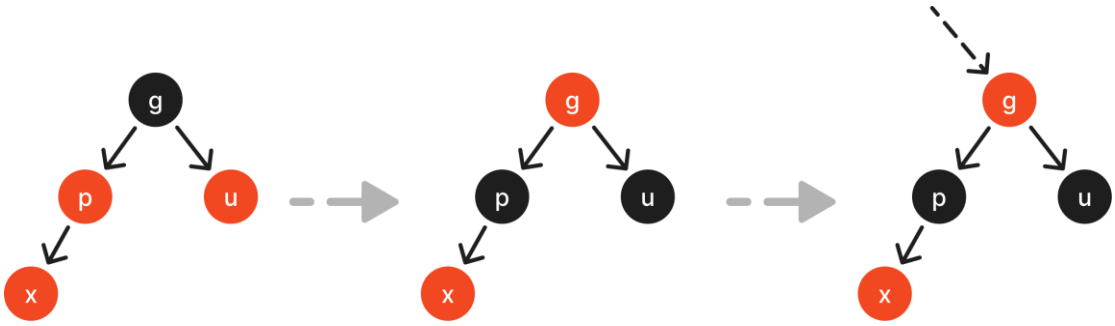
2.3 Red-Black Tree Maintenance

In this section, I will illustrate the typical operation of maintaining a binary search tree using insertion as an example. For the node x to be inserted, I define its parent node as p, its grandparent node as g, and its uncle node (i.e., the other child of the grandparent) as u.

Since a Red-Black Tree needs to maintain the same black height across all paths, the newly inserted node is generally marked as red. Therefore, adjustments will be required when the new red node is inserted, which can be categorized into the following four scenarios:

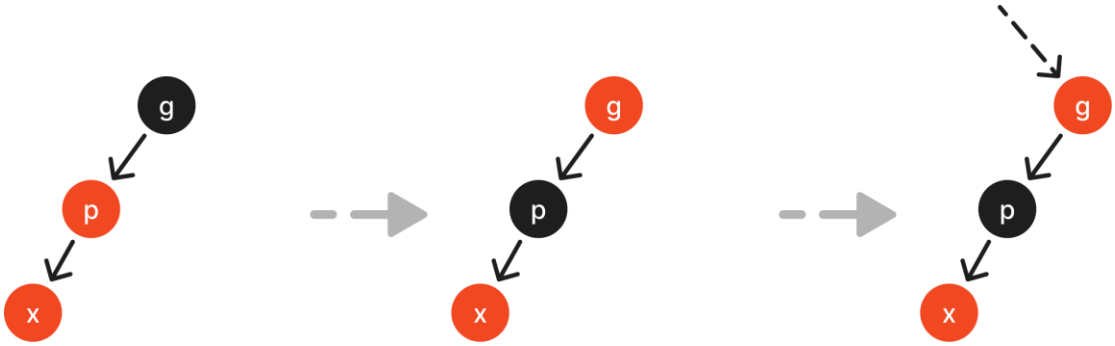
Scenario 1: Both Parent and Uncle are Red

1. Change the parent to black, thus accepting the new red node x. To maintain the local black height, also change the uncle node to black.
2. With the parent and uncle turned black, the black height changes, so the grandparent needs to be turned red to maintain overall black height consistency.
3. Treat the grandparent node as a newly inserted red node and recursively adjust upwards from the grandparent.



Scenario 2: Parent is Red, and Uncle Does Not Exist

1. The operation is the same as in Scenario 1, except the step to adjust the uncle node is omitted.

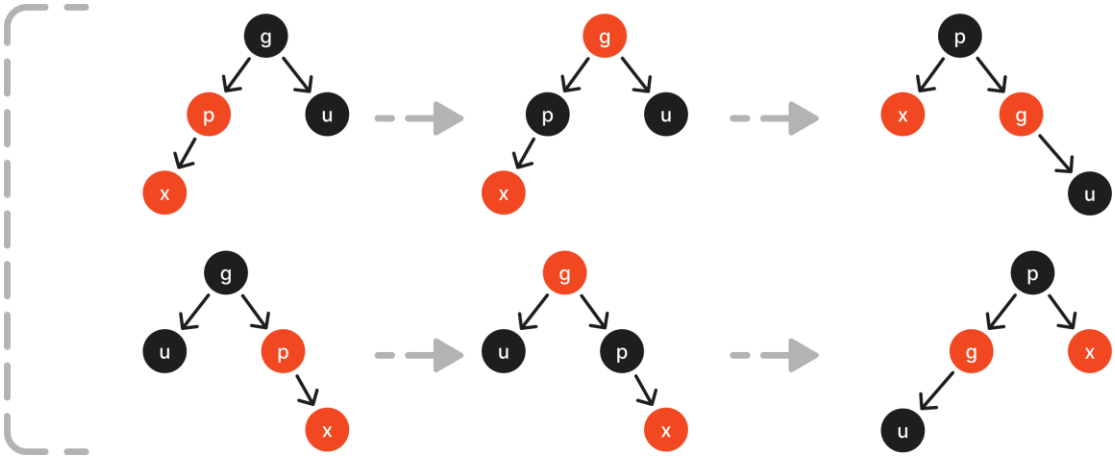


(In the above two scenarios, it makes no difference whether x *is a left child or a right child.)*

Scenario 3: Uncle is Black, and the New Node x and Parent p are Same-Direction Children

(Explanation: p is the left child of g and x is the left child of p, or p is the right child of g and x is the right child of p)

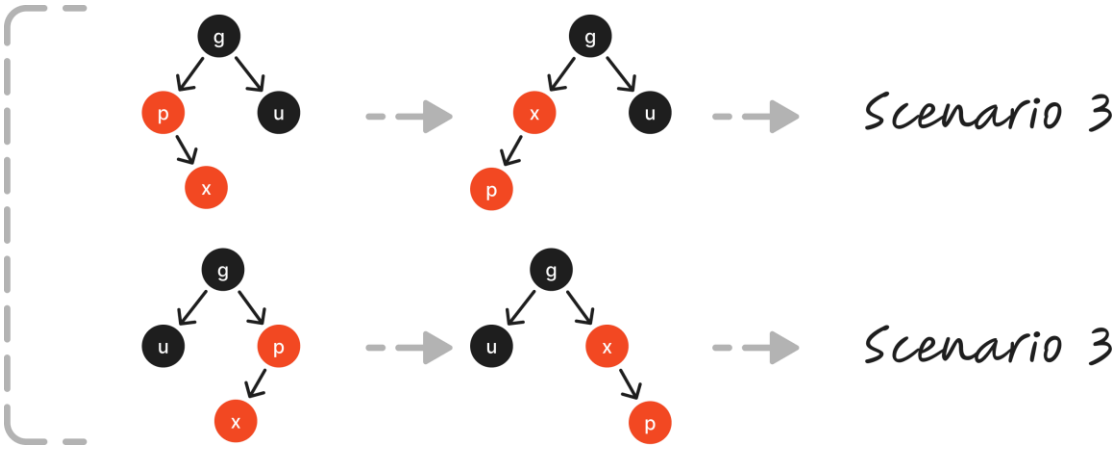
1. Change the parent to black and the grandparent to red (the black height of the right subtree is reduced).
2. Perform a right rotation on the grandparent, making the parent the new grandparent to restore the black height of the right subtree.
3. The top node becomes black again, maintaining the same state as before the insertion.



Scenario 4: Uncle is Black, and the New Node x and Parent p are Opposite-Direction Children

(Explanation: p is the left child of g and x is the right child of p, or p is the right child of g and x is the left child of p)

1. Make the parent the new x, perform a left rotation on the parent to create the initial state of Scenario 2.
2. Handle the new x (the original parent) according to Scenario 3.



By following these steps, the Red-Black Tree can be maintained efficiently, ensuring that the properties of the tree are preserved after each insertion.

2.4 Red-Black Tree Efficiency

The time complexity of search, insert, and delete operations in a Red-Black Tree is O(log n). During search operations, it performs similarly to a relatively balanced BST. However, with ordered data insertion, the Red-Black Tree's efficiency surpasses that of a BST, as the latter's time complexity deteriorates to O(n). For insert and delete operations, Red-Black Trees may require rotations and color changes, making them slightly less efficient than ordinary BSTs but maintaining an overall time complexity of O(log n).

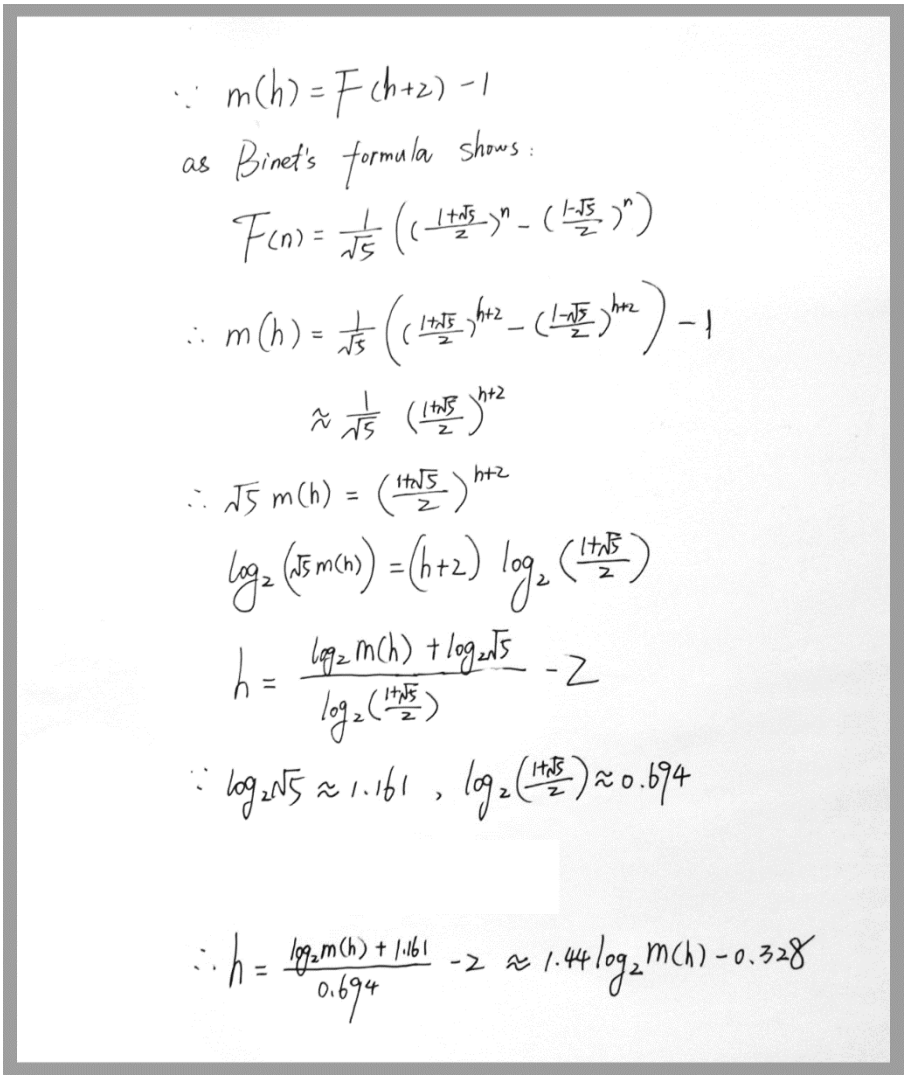
3.AVL Tree vs. Red-Black Tree

AVL Tree:

- Stricter balance criteria: The height difference between left and right subtrees does not exceed 1.
- Maximum height:To find the maximum height an AVL tree can achieve with a fixed number of nodes, we need to determine the minimum number of nodes $m(h)$ required to reach a given height h. The relationship $m(h)$ follows a recursive formula: $m(h)=m(h-1)+m(h-2)+1$, where $m(0)=1$ and $m(1)=2$. The three terms correspond to a subtree of height $h-1$, a subtree constrained by the balance factor to be at least height $h-2$, and a new root node. This recursive relation is very similar to the Fibonacci sequence, and by induction, we can derive that $m(h)=F(h+2)-1$, where $F(h)$ is the h-th Fibonacci number. By introducing the Binet's formula for Fibonacci numbers and simplifying, we obtain $1.44 * \log_2(m+2) - 1.328$. (The detailed derivation process can be

found in references 6 and 7.)

- Simplification process is shown below:



- For 1,000,000 nodes, the maximum height is 28.
- Search, insert, and delete operations have O(log n) complexity. Insertions require O(1) rotations, while deletions may need up to O(log n) rotations.

Red-Black Tree:

- Looser balance criteria: No path is more than twice as long as any other path.
- Maximum height: To ensure that the black height is equal, the longest path (alternating black and red nodes) in a red-black tree will not exceed twice the length of the shortest path (only black nodes). Therefore, the maximum height of the tree will be $2 * \log_2(n + 1)$
- For 1,000,000 nodes, the maximum height is 40
- Search, insert, and delete operations have O(log n) complexity. Both insertions and deletions require O(1) rotations.

Practical Performance:

- Although the time complexity of AVL Trees is slightly better than Red-Black Trees, modern CPUs are fast enough to make this performance difference negligible.
- In computer data structures, Red-Black Trees are more manageable for insertions and

deletions compared to AVL Trees.

- Overall, Red-Black Trees slightly outperform AVL Trees due to fewer rotations required.

References:

[1]Geeksforgeeks. Introduction to Red-Black Tree. https://www.geeksforgeeks.org/introduction-to-red-black-tree/

[2]S. Richard. Graph theory. http://www.math.nagoya-u.ac.jp/~richard/teaching/s2024/Graph.pdf, 2024.

[3]Wikipedia. Red–black tree — Wikipedia, the free encyclopedia.

[4]Wikipedia. AVL tree — Wikipedia, the free encyclopedia.

[5]Geeksforgeeks. AVL Tree Data Structure. https://www.geeksforgeeks.org/introduction-to-avl-tree/

[6]ICS-23. AVL. https://ics.uci.edu/~pattis/ICS-23/lectures/notes/AVL.txt

[7]StackOverFlow. Finding the minimum and maximum height in a AVL tree, given a number of nodes. https://stackoverflow.com/questions/30769383/finding-the-minimum-and-maximum-height-in-a-avl-tree-given-a-number-of-nodes