Analysis of Algorithms

BLG 335E

Project 3 Report

Mehmet Ali Balıkçı
balikci20@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 29.12.2023

1. Implementation

1.1. Differences between BST and RBT

Binary Search Trees (BSTs) and Red-Black Trees (RBTs) are both hierarchical data structures used to store data. However, RBTs are a type of self-balancing binary search tree that maintains a specific structure to ensure balance. Here are some distinctions between BSTs and RBTs:

BSTs are not inherently balanced, so if data is inserted without regard to balance, the tree's height can grow significantly. This can lead to inefficient performance for operations like insertion, deletion, and search.

In contrast, RBTs are self-balancing, ensuring that the tree's height remains O(log n), where n is the number of nodes. This guarantees efficient performance for all operations, as they have an upper bound of O(log n).

RBTs follow rules governing the color of each node, which maintain the tree's balance after insertions or deletions. These rules, such as coloring nodes red or black and avoiding adjacent red nodes, are crucial for the tree's balanced structure.

The rules for RBTs ensure that the tree maintains balance despite the initial shape or order of data insertion. This means that regardless of how the data is initially inserted, the height of the resulting RBT remains O(log n), providing consistent performance.

Different versions or distributions of data can impact the tree's height. Balanced data insertion results in a smaller tree height, while skewed insertion can lead to a taller tree in the case of BSTs. However, RBTs maintain a height of O(log n) due to their self-balancing nature, regardless of the data distribution.

In summary, RBTs guarantee balanced tree structure and efficient performance through their self-balancing rules, ensuring a consistent upper bound of O(log n) for all operations.

	Population1	**Population2**	**Population3**	**Population4**
RBT	21	24	24	16
BST	835	13806	12204	65

Table 1.1: Tree Height Comparison of RBT and BST on input data.

1.2. Maximum height of RBTrees

The maximum height of a Red-Black Tree containing n nodes is bounded by 2log2(n+1). This can be proven as follows:

In a general Binary Tree, let k be the minimum number of nodes on any root-to-NULL path. Then, it can be shown that n (the total number of nodes) is greater than or equal to 2^ k - 1. For example, if k is 3, then n is at least 7.

Operation	BST	RBT
Searching	O(h)	O(log n)
Deletion	O(h)	O(log n)
Insertion	O(h)	O(log n)
getHeight	O(n)	O(log n)
Inorder	O(n)	O(n)
Preorder	O(n)	O(n)
Postorder	O(n)	O(n)
Successor	O(h)	O(log n)
Predecessor	O(h)	O(log n)
findMaximum	O(h)	O(log n)
findMinimum	O(h)	O(log n)

For a Red-Black Tree, it can be proven that the minimum number of nodes on any root-to-NULL path is at most 2log2(n+1).

As a result, we have n >= 2^ (h/2) - 1, where h is the height of the tree.

Rearranging the terms, we obtain h <= 2log2(n+1).

Therefore, we conclude that the maximum height of a Red-Black Tree with n nodes is at most 2log2(n+1).

1.3. Time Complexity

Here's a summary table comparing the time complexity of operations for Binary Search Trees (BSTs) and Red-Black Trees (RBTs):

In the table, h represents the height of the tree, and n represents the number of nodes in the tree.

For BSTs, the worst-case scenario occurs when the tree is completely unbalanced, resulting in a height of n. In this case, operations such as searching, deletion, and insertion have a time complexity of O(n). In contrast, RBTs are self-balancing, so even in the worst-case scenario of complete imbalance (height 2log2(n+1)), these operations have a time complexity of O(log n).

Operations like getHeight, inorder, preorder, and postorder traversal have a time complexity of O(n) for both BSTs and RBTs because they require visiting every node in the tree.

Finding the maximum and minimum values in a BST or RBT has a time complexity of O(h), where h is the height of the tree. This is because traversing from the root to the leftmost or rightmost node allows finding the minimum or maximum value.

1.4. Brief Implementation Details

After adding the node during the insertion process in the RB tree, I check whether there is a problem with the color structure of the tree, and if there is a problem, I overcome this problem by performing rotation operations and, if necessary, color change operations. In the delete process, I check whether the rules of the tree are preserved after deleting

a node in the same way. If there is a problem, I solve it with rotation and, if necessary, color change. There are no such operations in BS tree. After deleting a node, I replace it with the smallest node from the tree on the right side of that node, so that the structure preserves itself.