Analysis of Algorithms

BLG 335E

Project 3 Report

KAAN KARATAŞ
karatask20@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 30.12.2023

1. Implementation

1.1. Differences between BST and RBT

Binary Search Tree and Red-Black Tree are both some type of self-balancing binary search trees. These two tree types differ mainly in how they maintain balance. In a BST, nodes are arranged based on key order, which might lead to uneven structures and slower performance in some cases. On the other hand, RBTs have specific rules, like color constraints and rotations, to keep the tree balanced during insertions and deletions.

BSTs organize nodes with lower values on the left and higher values on the right, creating a straightforward hierarchy. In contrast, RBTs, while still a type of BST, introduce a subtle complexity through five rules. These rules, involving color constraints and rotation mechanisms, ensure a balanced node arrangement. Despite this added complexity, RBTs offer enhanced stability during insertions and deletions compared to the more straightforward structure of BSTs.

The rules of Red-Black trees include:

1. **Root property** - The root is black.
2. **External property** - Every leaf (including nulls) is black.
3. **Internal property** - The children of a red node are black.
4. **Depth property** - All the leaves have the same black depth.
5. **Path property** - Every path from root to descendant leaf node contains same number of black nodes.

Maintaining a consistently shorter height than the Binary Search Tree, the Red-Black Tree owes its efficiency to its inherent self-balancing mechanism. This feature proves advantageous in operations like searching, calculating height, getting maximum and minimum values, as it significantly reduces the reliance on extensive recursive processes. The key significance lies in the tree's autonomous balancing, minimizing the need for lengthy recursive comparisons and enhancing the overall speed and efficiency of these operations.

This adherence to rules keeps the tree's height logarithmic, ensuring efficient operations. The advantage is clear when dealing with different data scenarios; RBTs maintain a more balanced structure, with a predictable height, compared to potentially

uneven BSTs. The set of rules in RBTs helps handle various data distributions, making them more reliable in different situations.

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

Table 1.1: Comparison of Heapsort and Quikcsort algorithms

In examining the data, it's clear that the Red-Black consistently maintains a significantly shorter height across different populations compared to the Binary Search Tree. For instance, in Population1, RBT achieves a height of 21, whereas BST has a much taller height of 835. This pattern continues in Population2 and Population3, where RBT maintains shorter heights (24) compared to BST (13806 and 12204, respectively). In Population4, RBT outperforms BST as well, with a height of 16 versus 65. These results underscore the efficiency of RBT's self-balancing mechanism, ensuring more streamlined operations and faster searches in various datasets.

1.2. Maximum height of RBTrees

The maximum height h of a Red-Black Tree with n nodes can be expressed as $(2 * log_2^{n+1})$. This outcome comes from the requirement that every path in an RBT that connects the root to a null pointer have an equal quantity of black nodes, which is one of the fundamental characteristics of Red-Black Trees.

Examine the case when a Red-Black Tree contains the most black nodes on any path in order to demonstrate this. In this instance, the tree is practically a balanced black-height tree. If n is the total number of nodes and $\lfloor x \rfloor$ is the floor function, then the maximum number of black nodes on any path is $\lfloor n/2 \rfloor$.

The relationship between a binary tree's number of leaves l and nodes n is represented by the equation $l = n + 1$. Every leaf on a red-black tree is seen as black. As a result, the maximum height of the tree can be written as $\lfloor n/2 + 1$ and h can be obtained by $h = \lfloor (l - 1/2 \rfloor * 1$ using the relationship between n and l.

Finally, by simplifying the expression, we arrive at the result $h = 2 * log_2^{n+1}$. This demonstrates that the Red-Black Tree maintains a balanced structure, ensuring that the maximum height is logarithmic with respect to the number of nodes.

1.3. Time Complexity

Write big-o complexity for each operation (searching, deletion, insertion ...) of RBTree and BSTree that you have implemented. Explain the reason behind the complexities in worst-case scenarios. You can use table like in previous example.

1.3.1. Binary Search Tree

The balance of the binary search tree determines the time complexity of different operations. After all, faster operations correspond with a more balanced tree. When the tree is balanced, the typical scenario has a time complexity of O(log n), where n is the number of nodes, for operations like insertion, deletion, and search. In the worst scenario, these operations might become less efficient and decay to O(n) when the structure of the tree is unbalanced. Because each node must be visited, traversing procedures and obtaining particular nodes have a linear complexity of time as O(n). Thus, a Binary Search Trees must be balanced for maximum effectiveness.

Preorder Traverse

- Worst case time complexity: $O(n)$

Inorder Traverse

- Worst case time complexity: $O(n)$

Postorder Traverse

- Worst case time complexity: $O(n)$

Search

- Worst case time complexity: $O(n)$

Successor

- Worst case time complexity: $O(logn)$

Predecessor

- Worst case time complexity: $O(logn)$

Insertion

- Worst case time complexity: $O(n)$

Deletion

- Worst case time complexity: $O(n)$

Height

- Worst case time complexity: $O(n)$

Minimum

- Worst case time complexity: $O(n)$

Maximum

- Worst case time complexity: $O(n)$

Total Nodes

- Worst case time complexity: $O(1)$

1.3.2. Red-Black Tree

Basic operations such as insertion, deletion, and search are guaranteed to have effective average-case time complexities of O(log n) thanks to the Red-Black Tree. This efficiency is preserved by the self-balancing characteristic, which offers reliable performance even in situations when equilibrium is not achieved. Because traversing operations require visiting each node, resulting with a linear time complexity of O(n). The height of the tree, determining the highest and lowest weighted nodes, and calculating the total number of nodes are examples of height-related operations that have a linear time complexity of O(n).

Preorder Traverse

- Worst case time complexity: $O(n)$

Inorder Traverse

- Worst case time complexity: $O(n)$

Postorder Traverse

- Worst case time complexity: $O(n)$

Search

- Worst case time complexity: $O(n)$

Successor

- Worst case time complexity: $O(logn)$

Predecessor

- Worst case time complexity: $O(logn)$

Insertion

- Worst case time complexity: $O(n)$

Deletion

- Worst case time complexity: $O(n)$

Height

- Worst case time complexity: $O(n)$

Minimum

- Worst case time complexity: $O(n)$

Maximum

- Worst case time complexity: $O(n)$

Total Nodes

- Worst case time complexity: $O(1)$

1.4. Brief Implementation Details

To preserve the integrity of the Red-Black Tree, a newly added node that is initially colored red goes through the insertFix method during the insertion process. This technique is applied when a newly inserted node doesn't match the properties of the tree, requiring rotations and color changes to bring the tree back into harmony. When the insertion affects the Red-Black Tree's characteristics, the insertFix method comes in handy to make sure the tree stays in check and follows the rules.

The central logic incorporates an essential method called 'deleteFix' to protect the Red-Black Tree's unique characteristics. The 'deleteFix' function is used when removing a node can change the intrinsic features of the tree. In this function, the deleteFix method coordinates node rotations and recoloring to bring them into compliance with the updated program requirements. By carefully adjusting to the changing needs of the software, this method guarantees that the Red-Black Tree maintains its structural integrity during the dynamic process of node elimination.