Analysis of Algorithms

BLG 335E

Project 3 Report

Şafak Özkan Pala
palas21@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

BST and RBT are both binary tree. They both store the data according to rules of binary search trees which is right node is bigger and left node is smaller than the current node. The main difference of these two tree is their insertion methods. BST inserts new node by adding it into the correct nil node while not changing anthing about the tree structure. However, RBT inserts new node by adding it into the correect nil node with its color. After insertion it will readjust the tree structure to maintain the RBT structure.

Red-Black Tree has 4 properties:

- Nodes are either red or black.
- The root and leaf are black.
- No two red nodes can be adjacent
- Every path from the root to any of the leaf nodes must have the same number of black nodes.

These properties lead tree to rearrange for the minimizing the height. These tree structure can be beneficial with several scenarios as it can be seen in the Table 1.1, RBT ensures a more consistent and predictable height across various input orders, in contrast to BST.

	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.

The population data in the ’population2.csv’ file is arranged in reverse order. Hence, BST created a tree with only left childs resulting height equals its nodes. This type of results are can be reduce the effectiveness of the binary search trees. Therefore, a solid algorithm for balancing the tree is required.

1.2. Maximum height of RBTrees

Proof:

- Let's merge all red nodes into to their parent black nodes.
- Thus, 2-3-4 tree is obtained with h' uniform depth of leaves.
- $h' \geq h/2$, since at most half of the leaves on any path are red.
- The number of leaves in each tree is $n + 1$
- $n + 1 \leq 2^{h'}$
- $\log(n + 1) \geq h' \geq h/2$
- $h \leq 2\log(n + 1)$

Therefore we can say that maximum height is $h \leq 2\log(n + 1)$

1.3. Time Complexity

Time complexities for given functions can be seen in the table below.

Operation	Red-Black Tree (RBT)	Binary Search Tree (BST)
Preorder Traversal	$O(n)$	$O(n)$
Inorder Traversal	$O(n)$	$O(n)$
Postorder Traversal	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(h)$
Successor and Predecessor	$O(\log n)$	$O(h)$
Insert	$O(\log n)$	$O(h)$
Delete Node	$O(\log n)$	$O(h)$
Get Height	$O(\log n)$	$O(h)$
Get Maximum and Get Minimum	$O(\log n)$	$O(h)$
Get Total Nodes	$O(n)$	$O(n)$

Note: n is the number of nodes, and h is the height of the tree.

1.3.1. BST

1. **Preorder Traversal:**
   - **Complexity:** $O(n)$ (visiting each node once).
   - **Proof:** Every node is visited exactly once in a preorder traversal.
2. **Inorder Traversal:**

- **Complexity:** $O(n)$ (visiting each node once).
- **Proof:** Every node is visited exactly once in an inorder traversal.

3. **Postorder Traversal:**
   - **Complexity:** $O(n)$ (visiting each node once).
   - **Proof:** Every node is visited exactly once in a postorder traversal.
4. **Search Tree:**
   - **Complexity:** $O(h)$ in the worst case, where h is the height of the tree.
   - **Proof:** Going from root to leaf to find the searching there for it is the height of the tree. In the worst case, a BST may degrade into a linked list, and the search time depends on the height.
5. **Successor and Predecessor:**
   - **Complexity:** $O(h)$ in the worst case, where h is the height of the tree.
   - **Proof:** Finding the successor or predecessor involves traversing from the root to the target node or vice versa.
6. **Insert:**
   - **Complexity:** $O(h)$ in the worst case, where h is the height of the tree.
   - **Proof:** In the worst case, the tree may be a single path, and insertion requires traversing from the root to a leaf.
7. **Delete Node:**
   - **Complexity:** $O(h)$ in the worst case, where h is the height of the tree.
   - **Proof:** In the worst case, the tree may be a single path, and deletion requires traversing from the root to a leaf.
8. **Get Height:**
   - **Complexity:** $O(n)$ in the worst case, as it involves visiting every node.
   - **Proof:** The height is determined by traversing the entire tree.
9. **Get Maximum and Get Minimum:**
   - **Complexity:** $O(h)$ in the worst case, where h is the height of the tree.
   - **Proof:** Max and min nodes are at the leaf.
10. **Get Total Nodes:**
    - **Complexity:** $O(n)$ in the worst case, as it involves visiting every node.
    - **Proof:** Counting every node requires traversing the entire tree.

1.3.2. RBT

1. **Preorder Traversal:**
    - **Complexity:** $O(n)$ (visiting each node once).
    - **Proof:** Every node is visited exactly once in a preorder traversal.
2. **Inorder Traversal:**
    - **Complexity:** $O(n)$ (visiting each node once).
    - **Proof:** Every node is visited exactly once in an inorder traversal.
3. **Postorder Traversal:**
    - **Complexity:** $O(n)$ (visiting each node once).
    - **Proof:** Every node is visited exactly once in a postorder traversal.
4. **Search Tree:**
    - **Complexity:** $O(\log n)$ in the worst case, where n is the number of nodes.
    - **Proof:** The self-balancing properties of Red-Black Trees ensure a logarithmic height, maintaining efficient search operations.
5. **Successor and Predecessor:**
    - **Complexity:** $O(\log n)$ in the worst case
    - **Proof:** The height of the Red-Black Tree is logarithmic, making successor and predecessor operations efficient.
6. **Insert:**
    - **Complexity:** $O(\log n)$ in the worst case
    - **Proof:** The self-balancing properties of Red-Black Trees ensure a logarithmic height, maintaining efficiency during insertion. After insertion fixup operation is called which is also has $O(\log n)$ complexity. Because these two functions need to iterate through root to leaf and leaf to root.
7. **Delete Node:**
    - **Complexity:** $O(\log n)$ in the worst case, where n is the number of nodes.
    - **Proof:** The self-balancing properties of Red-Black Trees ensure a logarithmic height, maintaining efficiency during deletion. After deletion there must be fixup like insertion so it does not effect the complexity of delete function.
8. **Get Height:**

- **Complexity:** $O(\log n)$ in the worst case, where n is the number of nodes.
- **Proof:** The self-balancing properties of Red-Black Trees ensure a logarithmic height, making height calculation efficient.

9. **Get Maximum and Get Minimum:**
   - **Complexity:** $O(\log n)$ in the worst case, where n is the number of nodes.
   - **Proof:** The self-balancing properties of Red-Black Trees ensure a logarithmic height, making finding the maximum and minimum efficient.
10. **Get Total Nodes:**
    - **Complexity:** $O(n)$ in the worst case, as it involves visiting every node.
    - **Proof:** Counting every node requires traversing the entire tree.

1.4. Brief Implementation Details

In this implementation of RBT, all of its properties are upheld. Two main function of RBT, which are deletion and insertion functions are implemented as follows. In `insert` function a new node created with the given parameters. After its creation, a pointer search for its place in the tree to fit the BST rules by comparing its value with the current nodes. When it is been found new node is placed into its correct place. After that `insertFix` function is invoked with created node. `insertFix` function will correct the colors and rearrange the colors according to RBT properties. It will examine the colors of relative nodes and according to their color it will perform left and right rotation. This will be continue untill the tree is fixed. After all operations root will be set black to guarantee that the root is black.

In `deleteNode` function we found the node with the `searchTree` function. After finding the node, we call the `deleteGivenNode` function with the founded node. In this function we consider three cases:

1. The node to be deleted has no or one child: It uses the `transplant` method to replace the subtree rooted at the node with the subtree rooted at its child (if any).
2. The node to be deleted has two children: It finds the successor (or predecessor) of the node, replaces the node with its successor (or predecessor), and updates the tree accordingly.
3. After deletion, it checks the color of the original node (`y`) and calls `deleteFix` only if the original color was black.

After deletion, the `deleteFix` method is called to restore Red-Black Tree properties. It considers cases where the sibling (`w`) of the deleted node (`x`) and its children have different colors, and performs rotations and color adjustments accordingly.The while loop iteratively moves up the tree, fixing violations until the properties are satisfied.