Analysis of Algorithms

BLG 335E

Project 3 Report

Abdullah Jafar Mansour Shamout
shamout21@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

Balancing

- **BST:** In a basic BST, the height of the tree can become skewed, leading to a worst-case time complexity of $O(n)$ for search, insert, and delete operations.
- **RBT:** Red-Black Trees are self-balancing. They maintain balance by enforcing additional properties, ensuring that the height of the tree remains logarithmic in relation to the number of nodes. This guarantees a worst-case time complexity of $O(\log n)$ for basic operations.

Node structures

- **BST:** Nodes in a BST have two childs, a parent, their value and name.
- **RBT:** Nodes in a Red-Black Tree have additional an attribute. A color (red or black). This extra information is used to maintain the balance of the tree.

Coloring Rules

- **RBT:** Red-Black Trees have specific rules about coloring to maintain balance:
  - Every node is either red or black.
  - The root is always black.
  - All leaves (null or NIL nodes) are considered black.
  - If a red node has children, both children are black.
  - Every path from a node to its descendant leaves has the same number of black nodes, ensuring that the tree remains balanced.

Rotations:

- **RBT:** Red-Black Trees may require tree rotations during insertion and deletion operations to maintain the properties of the tree. These rotations help to balance the tree while preserving the order.

Complexity:

- **BST:** In the worst case, the height of a basic BST can be n, where n is the number of nodes, leading to $O(n)$ complexity for basic operations.

- **RBT:** The self-balancing property of Red-Black Trees ensures that the height remains logarithmic, resulting in $O(\log n)$ time complexity for search, insert, and delete 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.

In the population1.csv file, the Red-Black Tree (RBT) has a height of 21, while the Binary Search Tree (BST) has a much higher height at 835. The population1.csv data is kinda jumbled, so the BST height isn't as bad as it is in population2.csv and population3.csv. In those files, the data is almost in order, making the BST stack up values heavily on one side, leading to a really tall tree. On the flip side, the RBT's height doesn't change much for population2.csv and population3.csv.

For the population4.csv file, where the data is kinda mixed up, the BST structure ends up almost as tall as the RBT. So, to sum it up, we can say that how well BSTs work depends on the order you give them data, while Red-Black Trees don't care much about the order; they stay efficient either way.

1.2. Maximum height of RBTrees

Let h be the height of the Red-Black Tree with n nodes.

1. Number of Black Nodes on Any Path:
    - In a Red-Black Tree, every path from the root to a null (NIL) pointer contains the same number of black nodes. Let b denote this number.
2. Minimum Number of Nodes on a Path:
    - The minimum number of nodes on any path from the root to a null pointer is $b + 1$, where b is the number of black nodes.
3. Maximum Height in Terms of Black Nodes:
    - The maximum height of a Red-Black Tree is obtained when every path from the root to a null pointer has the minimum number of nodes. Therefore, h is directly related to the number of black nodes (b).
4. Number of Nodes in Terms of Black Nodes:
    - The number of nodes (n) is related to the number of black nodes (b) by the inequality $n \geq 2^b - 1$. This is because, in the worst case, the Red-Black Tree is a perfect binary tree where each black node has two children, and the total number of nodes is $2^b - 1$.

5. Solving for b:
    - Solving the inequality $n \geq 2^b - 1$ for b gives $b \leq \log_2(n+1)$.
6. Maximum Height in Terms of Nodes:
    - The maximum height h is then $2 \cdot b$ (as each black node is followed by a red node or a leaf).
7. Substituting b:
    - Substituting the value of b into the maximum height formula, we get $h \leq 2 \cdot \log_2(n+1)$.

Therefore, the maximum height of a Red-Black Tree with n nodes is $2 \cdot \log_2(n+1)$. Which is $O(\log n)$

1.3. Time Complexity

Operation	Red-Black Tree (RBTree)	Binary Search Tree (BSTree)
Search	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(n)$
Minimum	$O(\log n)$	$O(n)$
Maximum	$O(\log n)$	$O(n)$
Successor	$O(\log n)$	$O(n)$
Predecessor	$O(\log n)$	$O(n)$
Inorder Traversal	$O(n)$	$O(n)$
Preorder Traversal	$O(n)$	$O(n)$
Postorder Traversal	$O(n)$	$O(n)$
Get Height	$O(n)$	$O(n)$
Get Total Nodes	$O(n)$	$O(n)$
Destructor	$O(n)$	$O(n)$

Table 1.2: Big-O Complexities of RBTree and BSTree Operations

Complexity Explanation

1. **Search:** Searching in Red-Black Tree takes $O(\log n)$ time as the tree is balanced, ensuring a logarithmic height. In contrast, Binary Search Tree's search has a worst-case complexity of $O(n)$ when the tree is skewed.
2. **Insertion:** Red-Black Tree maintains balance during insertion, resulting in $O(\log n)$ complexity. In Binary Search Tree, if the tree is skewed, the insertion can take $O(n)$ time.
3. **Deletion:** Deletion in Red-Black Tree is $O(\log n)$ due to its balanced nature. In Binary Search Tree, the worst-case scenario is $O(n)$ if the tree is skewed.

4. **Minimum/Maximum:** Finding the minimum or maximum in Red-Black Tree takes $O(\log n)$ due to its balanced structure. Binary Search Tree can take $O(n)$ in the worst case if it is skewed.

5. **Successor/Predecessor:** Red-Black Tree ensures $O(\log n)$ time for finding the successor or predecessor. In Binary Search Tree, it can take $O(n)$ if the tree is skewed.

6. **Inorder Traversal:** Both Red-Black Tree and Binary Search Tree take $O(n)$ for inorder traversal since it visits all nodes.

7. **Preorder/Postorder Traversal:** Traversals in both trees take $O(n)$ as each node is visited once.

8. **Get Height:** Getting the height in Red-Black Tree and Binary Search Tree is $O(n)$ as it requires traversing the entire height of the tree.

9. **Get Total Nodes:** Counting total nodes in Red-Black Tree and Binary Search Tree is $O(n)$ as all nodes need to be counted.

10. **Destructor:** The destructor in both trees has a complexity of $O(n)$ as it needs to free memory for all nodes.

1.4. Brief Implementation Details

General Remarks on Implementations

Nearly all functions are the same for both BST and RBT. The only difference was in handeling insertion and deletion where some fixing of the structure is needed. I also added destructors that free up memory once the code is done running

1.4.1. Red-Black Tree (RBT)

The Red-Black Tree (RBT) implementation ensures that it satisfies the rules through its operations by carefully handling the coloring and restructuring of nodes during insertion and deletion. Specifically:

1. **Insertion Operation:** When a node is inserted, the RBT algorithm ensures that the Red-Black properties are maintained. After the standard BST insertion, color adjustments and rotations are performed as needed to guarantee that no two consecutive red nodes exist on any path, and the black height property is maintained. Detailed explanation of each case and the way they are handled can be seen as comments throughout my code. But mainly, I have in insertBalance function that calls left and right rotation functions and does recoloring dependent on the case that the tree reaches after the insertion as taught in class.

2. **Deletion Operation:** During deletion, the RBT implementation covers various cases to preserve the Red-Black properties. Cases involving the deletion of a node with one or two children are considered, and appropriate rotations and recoloring are applied to maintain the balance and properties of the tree. Detailed explanation of each case and the way they are handled can be seen as comments throughout my code. But mainly, I have a deleteNodeBackend function that does transplants, and after its done it calls on the deleteBalance function that calls left and right rotation functions and does recoloring dependent on the case that the tree reaches after the deletion occurs as taught in class.

1.4.2. Binary Search Tree (BST)

In the Binary Search Tree (BST) implementation, the focus is on preserving the binary search tree property during deletion. This involves considering different cases based on the number of children of the node being deleted:

1. **Deletion with No Children:** If the node to be deleted has no children, it is simply removed.
2. **Deletion with One Child:** When a node has one child, the parent of the node is connected directly to its only child.
3. **Deletion with Two Children:** If a node has two children, the algorithm finds the node's in-order successor (or predecessor), replaces the node's value with the successor's (or predecessor's) value, and then recursively deletes the successor (or predecessor).

These strategies ensure that the binary search tree property is maintained after deletion operations in the BST.