Analysis of Algorithms

BLG 335E

Project 3 Report

Erblina Nivokazi
nivokazi21@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

While implementing Binary Search Trees (**BSTs**) and Red-Black Trees (**RBTs**), I noticed that the fundamental difference between them lies in their structural constraints. **BSTs**, even though they are simpler to implement, they lack rules to ensure a balanced structure. The shape of a **BST** is heavily influenced by the order of key insertions, which could potentially lead to a one-sided tree with a really high height in the worst-case scenario.

On the other hand, **RBTs** have a set of rules to maintain balance. Nodes in an **RBT** are colored red or black, and the rotations are performed to satisfy the specific properties and preserve the wanted structure. The enforced rules include; ensuring that the red nodes only have black children, the root and leaves (sentinels) are black, and every path from a node to its descendant leaves contains the same number of black nodes. These constraints guarantee a balanced height and provide a predictable structure, preventing the worst case scenario in which the tree form becomes highly skewed .

The importance of these implementations becomes apparent with different type of entries to the tree. To think that the values being put in the tree structure could be sorted, resulting in skewed data. **BSTs** would struggle with such scenarios, since in **BST** insertion does not fix the form of the tree it only adds to it, this would result in a dis-balanced tree with worst-case scenario height, and it would make searching, adding, and deleting in the tree very inefficient. Whereas, **RBTs** handle balanced and skewed data effectively, maintaining a logarithmic height.

Implementation-wise, **BSTs** are much simpler to implement, as they do not contain the numerous rules that **RBTs** do. **BSTs** check for **nullptr** suffices, the **BSTs** nodes to parent, right(child), and left(child) can all be compared with a **nullptr**. Whereas, **RBTs** require additional checks and rotations to maintain color properties. The usage of sentinels for leaf nodes simplifies boundary conditions in **RBT** implementation.

While implementing these structures, another thing I noticed and found a bit difficult to do was writing implementation code for insert and delete, fix helper functions were needed. To support and carry on the **RBT** structure functions, the sheer maintenance of the Red-Black Tree shape results in writing more functions and writing more complicated code.

BSTs offer simplicity at the cost of potential imbalance, while **RBTs**, with the enforced rules, ensure a balanced and efficient structure, which is a bit trickier to implement(or so I found). Sentinels play a crucial role in simplifying the implementation of boundary conditions in **RBTs**. Sentinels act as a standardized representation for leaf nodes, eliminating the need for explicit checks for **nullptr** conditions in the code. This consistent representation simplifies insertion, deletion, and rotation operations by providing a good placeholder for external leaves. The advantages of **RBTs** become more apparent in scenarios where a balanced structure and a good time-complexity are essential.

	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 structure of **RBTs** ensures a tree with optimal height and never skewed. As a result, from the findings from the C++ implementation of this homework, a difference in height can be seen between the two structures. **Red-Black Trees** have a more optimal height value due to their structure-preserving rules, whereas the **Binary Search Trees** can be seen to have visibly larger height values. To deeper analyze these results, we can look at the population files themselves. For the **population1.csv** file, the **RBT** has a height of 21, but **BST** has a height of value 835. The file **population1.csv** is moderately unordered, therefore the **BST** height value is not as bad as it is in **population2.csv** and **population3.csv**. In the second and third population csv files the data is almost ordered, in which case the **BST** keeps inserting values with heavy concentration in one side of the tree, thus resulting in a very big height value. In contrast, the height of **RBT** had little to almost no change in value for **population2.csv** and **population3.csv**. In the **population4.csv** file, the data is almost unordered, resulting in a **BST** structure that is very close to having the same height size as **RBT**. To conclude, we could say that the optimization of **BSTs** depends on the order of how input is provided, whereas the optimization of **RBTs** is independent of the order of input.

1.2. Maximum height of RBTrees

The maximum height of a Red-Black Tree (**RBT**) with n nodes is logarithmic with respect to the number of nodes, specifically $O(\log n)$. This logarithmic height is actually the reason why Red-Black Trees are considered so efficient and preferred for use.

1.2.1. Proof:

To prove this we have to examine the properties that the Red-Black Trees must satisfy:

1. Nodes colored Red cannot have red children. This constraint prevents long chains of red nodes, which contributes to the balanced structure of **RBTs**.
2. To reach any external leaf or sentinel in the tree, the path taken should have the same number of black nodes with each other path that must be taken to reach the other external leafs. This property ensures balanced paths throughout the tree.

To further analyze how this path is undertaken and how it affects the height of the tree, we can try to create the longest path in the Red-Black Tree. This path has red and black nodes consequently put beside each other due to the second property. Since the number of black nodes should be the same in each path, the longest height in a Red-Black Tree can be only double the length of the shortest path. So, the worst case for Red-Black Trees is not even close to the worst case of **BSTs** for which the worst case can be $O(\log n)$.

Let h be the height of the tree. The number of black nodes along the shortest path is $\lfloor h/2 \rfloor$. Thus, the maximum number of nodes n in the tree is given by:

$$n \geq 2^{\lfloor h/2 \rfloor}$$

Taking the logarithm base 2 of both sides:

$$\log_2 n \geq \lfloor h/2 \rfloor$$

Therefore, the height h is $O(\log n)$, making the Red-Black Tree a balanced data structure.

1.3. Time Complexity

Binary Search Tree Function Analysis

1. **Insertion (`insert` function):**

   - Time Complexity: $O(\log n)$ in the average case (balanced tree), $O(n)$ in the worst case (skewed tree).
   - Explanation: The insertion involves calling the helper function `insertHelper`, which has a time complexity that depends on the structure of the tree. So it is $O(\log n)$ when the tree is balanced and $O(n)$ when it is skewed.

2. **Search (`searchTree` function):**

   - Time Complexity: $O(\log n)$ in the average case, $O(n)$ in the worst case.
   - Explanation: The search involves calling the helper function `searchTreeHelper`, which has a time complexity that depends on the structure of the tree. So it is $O(\log n)$ when the tree is balanced and $O(n)$ when it is skewed.

3. **Deletion (`deleteNode` function):**

   - Time Complexity: $O(\log n)$ in the average case, $O(n)$ in the worst case.
   - Explanation: The deletion involves calling the helper function `deleteNodeHelper`, which has a time complexity that depends on the structure of the tree. So it is $O(\log n)$ when the tree is balanced and $O(n)$ when it is skewed.

4. **Successor (`successor` function):**
    - Time Complexity: $O(\log n)$ in the average case, $O(n)$ in the worst case.
    - Explanation: The successor function involves calling the helper function `successorHelper`, which traverses the right subtree.
5. **Predecessor (`predecessor` function):**
    - Time Complexity: $O(\log n)$ in the average case, $O(n)$ in the worst case.
    - Explanation: The predecessor function involves calling the helper function `predecessorHelper`, which traverses the left subtree.
6. **Inorder (`inorder` function):**
    - Time Complexity: $O(n)$.
    - Explanation: The inorder traversal involves calling the helper function `inOrderHelper` for each node, visiting each node once.
7. **Preorder (`preorder` function):**
    - Time Complexity: $O(n)$.
    - Explanation: The preorder traversal involves calling the helper function `preOrderHelper` for each node, visiting each node once.
8. **Postorder (`postorder` function):**
    - Time Complexity: $O(n)$.
    - Explanation: The postorder traversal involves calling the helper function `postOrderHelper` for each node, visiting each node once.
9. **Get Height (`getHeight` function):**
    - Time Complexity: $O(n)$.
    - Explanation: The `getHeight` function involves calling the helper function `getHeightHelper` for each node, visiting each node once.
10. **Get Maximum (`getMaximum` function):**
    - Time Complexity: $O(\log n)$ in the average case, $O(n)$ in the worst case.
    - Explanation: The function traverses the right subtree.
11. **Get Minimum (`getMinimum` function):**
    - Time Complexity: $O(\log n)$ in the average case, $O(n)$ in the worst case.
    - Explanation: The function traverses the left subtree.

12. **Get Total Nodes (`getTotalNodes` function):**
    - Time Complexity: $O(n)$.
    - Explanation: The `getTotalNodes` function involves calling the helper function `totalNodesHelper` for each node, visiting each node once.

Red-Black Tree Function Analysis

1. **Insert (`insert` Function):**
    - Time Complexity: $O(\log n)$.
    - Explanation: The insertion operation in a Red-Black Tree involves finding the appropriate position for the new node and then fixing the tree to maintain Red-Black Tree properties. The height of a Red-Black Tree is logarithmic in the number of nodes, so the insert operation takes $O(\log n)$ time.
2. **Delete (`deleteNode` Function):**
    - Time Complexity: $O(\log n)$.
    - Explanation: Similar to the `insert` operation, the delete operation involves finding and deleting a specific node, followed by fixing the Red-Black Tree properties. The height of the Red-Black Tree is logarithmic, so the delete operation takes $O(\log n)$ time.
3. **Search (`searchTree` Function):**
    - Time Complexity: $O(\log n)$.
    - Explanation: Searching for a node in a Red-Black Tree is similar to searching in a Binary Search Tree. The time complexity is $O(\log n)$ since it depends on the height of the tree.
4. `successor` **and** `predecessor` **Functions:**
    - Time Complexity: $O(\log n)$.
    - Explanation: Finding the successor or predecessor involves traversing the tree from the current node. In a balanced tree like Red-Black Tree, the height is logarithmic, resulting in $O(\log n)$ time complexity.
5. `getHeight` **Function:**
    - Time Complexity: $O(n)$.
    - Explanation: The `getHeight` function traverses the tree to find the maximum height. In the worst case, it visits every node once, resulting in a time complexity of $O(n)$.

6. `getTotalNodes` **Function:**
   - Time Complexity: $O(n)$.
   - Explanation: Similar to `getHeight`, the `getTotalNodes` function traverses the entire tree to count the number of nodes. It also has a time complexity of $O(n)$.

7. `getMaximum` **and** `getMinimum` **Functions:**
   - Time Complexity: $O(\log n)$.
   - Explanation: The function traverses the tree, since in Red-Black Trees the height is stable $O(\log n)$, these functions traverse for $O(\log n)$ times.

8. `preorder`**,** `inorder` **and** `postorder` **Functions:**
   - Time Complexity: $O(n)$.
   - Explanation: These functions traverse the whole tree, so their time complexity will be the size of the tree, respectively $O(n)$.

The Red-Black Trees manage to maintain a balanced structure, providing more efficient search, insert, and delete operations with logarithmic time complexity. But still some functions that involve traversing the entire tree (like getHeight and getTotalNodes) will still have a linear time complexity.

1.4. Brief Implementation Details

1.4.1. Binary Search Tree

My implementation of the `Binary Search Tree` is made to manage or sort cities by their population value. The primary class, `BinarySearchTree`, is responsible for handling various operations on the BST.

1. **Insertion Operation:**

   The insertion operation is made by the `insert` method. It ensures the Binary Search Tree property by recursively traversing the tree, comparing data values, and appropriately placing the new node to the left or right.

2. **Deletion Operation:**

   Deletion is handled by the `deleteNode` method. To preserve the Binary Search Tree properties, the implementation considers three cases:

   (a) Node has no left child: Replace with its right child.

   (b) Node has no right child: Replace with its left child.

   (c) Node has both left and right children: Find the successor, replace the current node's data with the successor's data, and recursively delete the successor.

3. **Search Operation:**

   The `searchTree` method is responsible for searching the tree for a specific population value. It traverses the tree and checks for nodes that have data equal to the target value.

4. **Traversal Operations:**

   The class provides three traversal methods: `inorder`, `preorder`, and `postorder`. These methods use recursive helper functions (`inOrderHelper`, `preOrderHelper`, and `postOrderHelper`) to traverse the tree and perform actions (e.g., printing) at each node.

5. **Height Calculation:**

   The `getHeight` method calculates the height of the tree using a recursive helper function (`getHeightHelper`). It considers the maximum height between the left and right subtrees.

6. **Node Retrieval:**

   Methods `getMaximum` and `getMinimum` retrieve the nodes with the maximum and minimum values, respectively. They traverse the tree to the rightmost and leftmost nodes, respectively.

7. **Additional Functions:**

   The implementation includes functions to get the total number of nodes (`getTotalNodes`) and to find the successor and predecessor of a given node (`successor` and `predecessor`).

1.4.2. Red-Black Tree

1. **Insertion Operation:**

   - The `insert` function follows the standard Red-Black Tree insertion algorithm.
   - The `insertHelper` function finds the appropriate position for the new node and ensures Red-Black Tree properties are maintained.
   - `insertFix` is invoked after insertion to handle cases where Red-Black Tree properties might be violated.

2. **Deletion Operation:**

   - The `deleteNode` function initiates deletion, calling `deleteNodeHelper`.

- `deleteNodeHelper` finds and handles the replacement and deletion process.
- `deleteFixUp` is invoked after deletion to preserve Red-Black Tree properties.

3. **Rotation Operations:**
   - `leftRotate` and `rightRotate` perform left and right rotations during insertion and deletion operations.
4. **Tree Traversal:**
   - `preorder`, `inorder`, and `postorder` functions facilitate tree traversal.
   - Corresponding helper functions (`preOrderHelper`, `inOrderHelper`, and `postOrderHelper`) are implemented recursively.
5. **Utility Functions:**
   - `searchTree`: Searches for a node with a specific key.
   - `successor` and `predecessor`: Find the successor and predecessor nodes, respectively.
   - `getHeight`: Determines the height of the Red-Black Tree.
   - `getMaximum` and `getMinimum`: Retrieve the maximum and minimum nodes in the tree.
   - `getTotalNodes`: Returns the total number of nodes in the Red-Black Tree.
6. **Sentinel Usage:**
   - A sentinel node represents null leaves, addressing the need for a non-null structure in Red-Black Tree operations.
7. **Color Coding:**
   - Nodes are assigned colors (0 for black, 1 for red) to satisfy Red-Black Tree properties.
   - Color adjustments are made during insertion and deletion processes to ensure these properties are maintained.
8. **Complexity Analysis:**
   - The implementation achieves efficient operations with time complexity proportional to the height of the tree.
   - Rotations and color adjustments help in balancing the tree and maintaining logarithmic height.