Analysis of Algorithms

BLG 335E

Project 3 Report

Yusuf Yıldız
yildizyu21@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 (BST) and Red-Black Trees (RBT) are both types of binary trees used for organizing and searching data. However, there are key differences between them, primarily stemming from the set of rules that define the structure and behavior of a Red-Black Tree.

	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.

Red-Black Tree Rules:

- **Node Colors:** Every node is colored, either red or black.
- **Root and Leaf Properties:**
  - The root is black.
  - Every leaf (`nullptr`) is black.
- **Red Node Constraints:** Red nodes cannot have red children.
- **Black Height:** Every path from a node to its descendant leaves contains the same number of black nodes, known as the black height.

Impact on Tree Structure:

The set of Red-Black Tree rules ensures a balanced structure, preventing the tree from becoming skewed. This balance is achieved by redistributing colors and adjusting the structure during insertions and deletions.

Advantages of Red-Black Trees:

- **Balanced Height:** Red-Black Trees maintain a balanced height, providing efficient search, insertion, and deletion operations with a logarithmic time complexity.
- **Predictable Performance:** The worst-case height of a Red-Black Tree is guaranteed to be logarithmic, leading to predictable performance in various scenarios.
- **Self-Balancing:** Red-Black Trees automatically adjust their structure during insertions and deletions, eliminating the need for manual rebalancing.

Detailed Discussion

- **1. Ordered Data**
  - **Population1 (Descending):** In Population1, where data is ordered in descending order, the Red-Black Tree (RBT) exhibits a height of 21, significantly outperforming the Binary Search Tree (BST) with a height of 835. This stark contrast underscores the ability of the RBT to maintain balance, preventing the tree from becoming skewed, and ensuring efficient operations.
  - **Population2 (Ascending):** Similarly, in Population2 (ascending order), the RBT height remains at 24, while the BST height is 13806, which is because it is built just like the linked list. This reinforces the consistent performance of the RBT in maintaining logarithmic height even in ordered datasets.
- **2. Partially Ordered Data**
  - **Population3 (Partially Ordered):** For Population3, where the data is partially ordered in ascending order, the RBT still maintains a height of 24, comparable to the fully ordered Population2. In contrast, the BST height is 12204. This highlights the adaptability of RBT to different degrees of ordering, showcasing its resilience.
- **3. Random Shuffle**
  - **Population4 (Random Shuffle):** In Population4, where data is shuffled randomly, the RBT performs exceptionally well with a height of 16. The BST, on the other hand, has a height of 65. BST's best performance is in this population because BST is dramatically affected by the input data order and randomness helps the building process of the BST. Even with a completely random arrangement, the RBT's self-balancing mechanisms efficiently handle the data, resulting in a more balanced structure.
- **4. Advantages of Red-Black Trees**
  - **Consistent Logarithmic Heights:** The Red-Black Tree consistently maintains logarithmic heights across different datasets. This ensures that search, insertion, and deletion operations have a predictable and efficient time complexity, providing a stable performance guarantee.
  - **Self-Balancing Mechanisms:** The self-balancing properties of the Red-Black Tree eliminate the need for manual rebalancing, making it a resilient data structure. Regardless of the data distribution, the tree adapts and optimizes its structure dynamically.

Conclusion

In summary, the experiment outcomes underscore the advantages of Red-Black Trees in maintaining balanced structures across a spectrum of data distributions. Whether the data is ordered, partially ordered, or randomly shuffled, the Red-Black Tree consistently exhibits logarithmic heights, ensuring efficient and predictable performance compared to Binary Search Trees. The self-balancing nature of Red-Black Trees makes them a robust choice for scenarios where data characteristics may vary.

1.2. Maximum Height of RBTrees

Proving the Height of Red-Black Trees

The height of a red-black tree is crucial for calculating its asymptotic complexity and performance. This proof outlines one method of determining the maximum height.

First, imagine a red-black tree with height h. Now, merge all red nodes into their black parents. A black node can have:

1. 2 black children, in which case the black parent still has 2 children.
2. 1 black child and 1 red child, in which case the black parent now has 3 children.
3. 2 red children, in which case the black parent now has 4 children.

Here is a graphical example of that merging process (assume any stray arrow points to a black node).

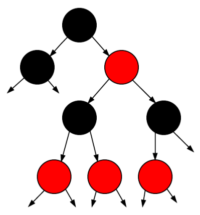


Figure 1.1: Merging red nodes into black parents.

Now, merge all red nodes into their parent nodes. This means that any black node will now have $2 \cdot r + 2$ pointers coming out of it, where r is the number of red children they have.

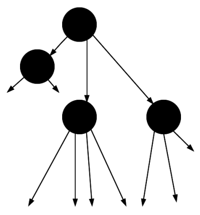


Figure 1.2: Merged tree with red nodes incorporated.

As you can see, every black node has either 2, 3, or 4 children.

This new tree has a height, h_1. Because any given path in the original red-black tree had at most half its nodes red, we know that this new height is at least half the original height. So,

$$h_1 \geq \frac{h}{2}$$

The number of leaves in a tree is exactly equal to $n + 1$, so

$$n + 1 \geq 2^{h_1}$$

Taking the logarithm base 2 of both sides:

$$\log_2(n + 1) \geq h_1 \geq \frac{h}{2}$$

Simplifying:

$$h \leq 2\log_2(n + 1)$$

1.3. Time Complexity

Big-O time complexities of each function in RBTree and BSTree are as the table below. For BSTree functions, h means the height of the tree, and it is highly dependent on the input data because BSTree has no balancing mechanism as the one in RBTree. So h may be the n at worst-case, which is the size of the input data, or lower than n depending on the input data order.

Operation	**RBT Time Complexity**	**BST Time Complexity**
Search	$O(\log n)$ (`searchTree`)	$O(h)$ (`searchTree`)
Insertion	$O(\log n)$ (`insert`)	$O(h)$ (`insert`)
Deletion	$O(\log n)$ (`deleteNode`)	$O(h)$ (`deleteNode`)
Traversal	$O(n)$ (`preorder,inorder,postorder`)	$O(n)$ (`preorder,inorder,postorder`)
Get Height	$O(n)$ (`getHeight`)	$O(n)$ (`getHeight`)
Successor/Predecessor	$O(\log n)$ (`successor, predecessor`)	$O(h)$ (`successor, predecessor`)
Max/Min	$O(\log n)$ (`getMaximum, getMinimum`)	$O(h)$ (`getMaximum, getMinimum`)
Total Nodes	$O(n)$ (`getTotalNodes`)	$O(n)$ (`getTotalNodes`)

Table 1.2: Time Complexity of RBT and BST Operations.

Time Complexity Analysis of BSTree

1. **Search Operation** (`searchTree`):
   - **Worst-case Time Complexity:** $O(h)$, where h is the height of the tree.
   - **Explanation:** In the worst case, the search operation traverses the height of the tree. For a balanced BST, the height is logarithmic ($O(\log n)$), but in the worst case (skewed tree), it becomes $O(n)$.
2. **Insertion Operation** (`insert`):
   - **Worst-case Time Complexity:** $O(h)$, where h is the height of the tree.
   - **Explanation:** Similar to the search operation, insertion involves finding the correct position for the new node, which requires traversing the height of the tree.
3. **Deletion Operation** (`deleteNode`):
   - **Worst-case Time Complexity:** $O(h)$, where h is the height of the tree.
   - **Explanation:** Deletion involves finding the node to delete ($O(h)$) and possibly transplanting a node (also $O(h)$). So, the worst-case time complexity is $O(h)$.
4. **Traversal Operations** (`preorder`, `inorder`, `postorder`):
   - **Worst-case Time Complexity:** $O(n)$, where n is the number of nodes in the tree.
   - **Explanation:** Each node needs to be visited once during traversal, so the time complexity is $O(n)$.
5. **Get Height Operation** (`getHeight`):
   - **Worst-case Time Complexity:** $O(n)$, where n is the number of nodes in the tree.
   - **Explanation:** The `getHeight` operation involves traversing the entire tree to find the height. In a balanced tree, this is $O(\log n)$, but in the worst case (skewed tree), it becomes $O(n)$.
6. **Successor and Predecessor Operations** (`successor` and `predecessor`):
   - **Worst-case Time Complexity:** $O(h)$, where h is the height of the tree.
   - **Explanation:** These operations involve traversing the height of the tree in certain cases.
7. **Get Maximum and Minimum Operations** (`getMaximum` and `getMinimum`):

- **Worst-case Time Complexity:** $O(h)$, where h is the height of the tree.
- **Explanation:** These operations involve traversing the height of the tree in the worst case.

8. **Get Total Nodes Operation** (`getTotalNodes`):

   - **Worst-case Time Complexity:** $O(n)$, where n is the number of nodes in the tree.
   - **Explanation:** The `getTotalNodes` operation involves counting all nodes in the tree, resulting in $O(n)$ time complexity.

Time Complexity Analysis of RBTree

1. **Search Operation (`searchTree`):**

   - **Worst-case Time Complexity:** $O(\log n)$
   - **Explanation:** Similar to the Binary Search Tree, the Red-Black Tree maintains the binary search property, resulting in logarithmic time complexity for the search operation.

2. **Insertion Operation (`insert`):**

   - **Worst-case Time Complexity:** $O(\log n)$
   - **Explanation:** The insertion operation involves finding the correct position for the new node and may require fixup operations to maintain the Red-Black Tree properties. The tree's height is logarithmic, resulting in $O(\log n)$ time complexity.

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

   - **Worst-case Time Complexity:** $O(\log n)$
   - **Explanation:** Similar to the insertion operation, deletion involves finding the node to delete and may require fixup operations. The height of the Red-Black Tree is logarithmic, resulting in $O(\log n)$ time complexity.

4. **Traversal Operations (`preorder`, `inorder`, `postorder`):**

   - **Worst-case Time Complexity:** $O(n)$
   - **Explanation:** Traversal operations visit each node once, resulting in linear time complexity, where n is the number of nodes in the tree.

5. **Get Height Operation (`getHeight`):**

   - **Worst-case Time Complexity:** $O(n)$

- **Explanation:** Similar to the Binary Search Tree, the `getHeight` operation traverses the entire tree. In a balanced tree, this is $O(\log n)$, but in the worst case (skewed tree), it becomes $O(n)$.

6. **Successor and Predecessor Operations (`successor` and `predecessor`):**
   - **Worst-case Time Complexity:** $O(\log n)$
   - **Explanation:** The successor and predecessor operations involve traversing the height of the tree in certain cases, resulting in $O(\log n)$ time complexity.
7. **Get Maximum and Minimum Operations (`getMaximum` and `getMinimum`):**
   - **Worst-case Time Complexity:** $O(\log n)$
   - **Explanation:** Finding the maximum and minimum values involves traversing the height of the tree in the worst case, resulting in $O(\log n)$ time complexity.
8. **Get Total Nodes Operation (`getTotalNodes`):**
   - **Worst-case Time Complexity:** $O(n)$
   - **Explanation:** The `getTotalNodes` operation involves counting all nodes in the tree, resulting in linear time complexity, where n is the number of nodes.

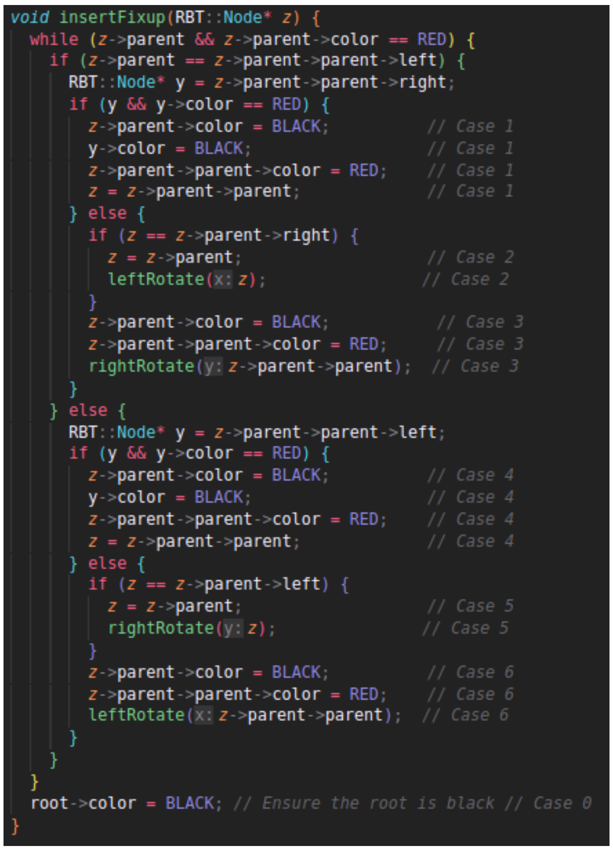
To conclude, it is obvious that the RBTree function's complexities converged to $O(\log n)$ by the balancing mechanism. On the contrary, the BSTree functions may be run in $O(n)$ at the worst case depending on the input data. RBTrees maintain a balanced height, providing efficient search, insertion, and deletion operations with guaranteed logarithmic time complexity in the worst case.

1.4. Brief Implementation Details

Red-Black Tree (RBT)

Ensuring RBT Rules:

- **Insertion Operation:**
  - During insertion, the Red-Black Tree properties are maintained. First, the correct position for the new node is searched and the necessary links are made. Then, the insertFixup function is called with the newly inserted node.
  - The `insertFixup` function corrects any violations by recoloring nodes and performing rotations. Which is shown as:
- **Deletion Operation:**



```
void insertFixup(RBT::Node* z) {
  while (z->parent && z->parent->color == RED) {
    if (z->parent == z->parent->parent->left) {
      RBT::Node* y = z->parent->parent->right;
      if (y && y->color == RED) {
        z->parent->color = BLACK;           // Case 1
        y->color = BLACK;                   // Case 1
        z->parent->parent->color = RED;     // Case 1
        z = z->parent->parent;              // Case 1
      } else {
        if (z == z->parent->right) {
          z = z->parent;                    // Case 2
          leftRotate(x: z);                // Case 2
        }
        z->parent->color = BLACK;            // Case 3
        z->parent->parent->color = RED;      // Case 3
        rightRotate(y: z->parent->parent);  // Case 3
      }
    } else {
      RBT::Node* y = z->parent->parent->left;
      if (y && y->color == RED) {
        z->parent->color = BLACK;           // Case 4
        y->color = BLACK;                   // Case 4
        z->parent->parent->color = RED;     // Case 4
        z = z->parent->parent;              // Case 4
      } else {
        if (z == z->parent->left) {
          z = z->parent;                    // Case 5
          rightRotate(y: z);               // Case 5
        }
        z->parent->color = BLACK;           // Case 6
        z->parent->parent->color = RED;     // Case 6
        leftRotate(x: z->parent->parent);  // Case 6
      }
    }
  }
  root->color = BLACK; // Ensure the root is black // Case 0
}
```

Figure 1.3: insertFixup function implementation.

- The `deleteNode` function preserves the Red-Black Tree properties. It handles various cases for each situation in which the node is to be deleted whether a right or left child of its parent. Then, the delerteFixup function is called with the last version of the node to be deleted.
- The `deleteFixup` function adjusts the tree, handling cases like double-black nodes, rotations, and recoloring. The figure of this function is not given due to its complexity.

Binary Search Tree (BST)

Handling Different Cases in Deletion:

The `deleteNode` function in BST handles cases to preserve the binary search tree property. Three main cases are considered based on the number of children the node to be deleted has.

- Case 1: Node has 0 children (Leaf Node):
  - The node is removed, and its parent's pointer is set to `nullptr`.
- Case 2: Node has 1 child:
  - The child node replaces the deleted node, and the parent's pointer is updated.

- Case 3: Node has 2 children:
  - The successor node (minimum value in the right subtree) replaces the deleted node.
  - If the successor has a right child, it takes the place of the successor in the tree. Then the successor's right child is linked to the right child of the node to delete. Finally, the node to delete is transplanted with the successor node.

  The `transplant` function replaces one subtree with another, ensuring correct pointer updates.

In summary, both the BST and RBT implementations address the preservation of the respective tree properties during insertion and deletion operations. The provided code incorporates necessary adjustments, rotations, and recoloring to ensure that the tree properties are maintained and the desired balance is achieved.