Analysis of Algorithms

BLG 335E

Project 3 Report

FEYZA NUR KELEŞ
kelesf22@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: DATE HERE

1. Implementation

Throughout this report, I aimed to keep my explanations clear and detailed while avoiding unnecessary complexity. In my past assignments, I lost points for both including too much detail and for being too vague in certain parts. With this report, I have tried to find a good balance that meets the requirements.

I chose not to include code snippets here since the code will be graded separately alongside this report. Instead, I focused on describing the logic and methodology behind the implementation, as well as analyzing the results. I hope this one aligns with the expectations.

1.1. Data Insertion

In this section, we construct and analyze two types of trees to store the dataset: a **Binary Search Tree (BST)** and a **Red-Black Tree (RBT)**. The trees are used to insert and manage the data while updating cumulative sales for publishers when duplicates occur. Additionally, we measure and compare the time taken to insert all the data into each tree.

1.1.1. Binary Search Tree (BST)

A Binary Search Tree follows a hierarchical structure where each node has:

- A left child containing nodes with keys (publisher names) smaller than the parent.
- A right child containing nodes with keys greater than the parent.

In this implementation:

- The publisher's name acts as the key to determine the position of the node.
- If a publisher already exists in the tree, instead of creating a new node, its sales data (North America, Europe, and Others) are updated cumulatively.

The insertion logic ensures that:

- Nodes are compared recursively until the correct position (left or right) is found.
- Duplicate publishers are detected by comparing their keys. If they match, the existing sales data is incremented.

The time taken for data insertion into the BST is recorded as:

$$\text{Time for BST: } 27986\,\mu s$$

In the worst case (e.g., when the data is already sorted), the height of the tree becomes linear $O(n)$, resulting in slower insertions.

1.1.2. Red-Black Tree (RBT)

A Red-Black Tree is a balanced binary search tree that maintains its height as $O(\log n)$, regardless of the input data. This balancing is achieved by enforcing the following properties:

- Each node is assigned a color: either **red** (1) or **black** (0).
- The root node is always black.
- No two consecutive red nodes are allowed (a red node cannot have a red parent or child).
- Every path from a node to its leaf nodes contains the same number of black nodes.

The insertion process in the RBT includes two main steps:

1. Insert the node as in a regular BST.
2. Fix any violations of the Red-Black properties using rotations and color changes (`RB_insert_fixup` function).

Key Steps in Insertion and Fixup:

- If a new node is added as the child of a red node, it may violate the property of consecutive red nodes. In such cases:
  - The uncle node's color is checked.
  - If the uncle is red, recoloring occurs to resolve the violation.
  - If the uncle is black, rotations (left or right) are performed to restore balance.
- The root is always recolored to black at the end of the process.

The time taken for data insertion into the RBT is recorded as:

Time for RBT: $25182\,\mu s$

1.1.3. Comparison of Results

Tree Type	Insertion Time (μs)
Binary Search Tree (BST)	27986
Red-Black Tree (RBT)	25182

Table 1.1: Insertion Times for BST and RBT

- The Red-Black Tree outperformed the Binary Search Tree in terms of insertion time. This is because the RBT maintains a balanced structure, ensuring logarithmic time complexity for insertions.

- In the BST, the lack of balancing may lead to a skewed tree, particularly if the dataset is sorted or nearly sorted. This results in higher insertion times as the tree height increases.
- The additional overhead in the RBT, such as rotations and recoloring, is minimal compared to the performance gains achieved by maintaining balance.

Both trees successfully stored the dataset while updating cumulative sales for duplicate publishers. However, the Red-Black Tree demonstrated better efficiency due to its balanced structure.

1.2. Search Efficiency

In this section, we perform randomized searches on both the Binary Search Tree (BST) and Red-Black Tree (RBT) to evaluate their search performance. The objective is to measure and compare the average search time taken to locate a random publisher from a list of publishers, using both tree structures.

1.2.1. Search Process in BST

The Binary Search Tree (BST) relies on the principle of binary search to locate a specific key, in this case, the publisher's name. The structure of the BST ensures that:

- For any given node, all keys in its left subtree are smaller, and all keys in its right subtree are larger.
- The search begins at the root and recursively moves down the tree, comparing the key at each node with the search key.
- If a match is found, the node is returned; otherwise, the search continues to the left or right subtree depending on whether the search key is smaller or larger than the current node's key.

The time complexity for searching in a BST depends on the height of the tree. In the best case, when the tree is perfectly balanced, the search time is $O(\log n)$, where n is the number of nodes. However, in the worst case, where the tree is skewed (i.e., it resembles a linked list), the time complexity degrades to $O(n)$.

Average Search Time for BST: 1377.5 ns

1.2.2. Search Process in RBT

Overall the search algorithm is the same but unlike the BST, the RBT's height is guaranteed to be logarithmic, even in the worst case. As a result, the search operation in an RBT is expected to have a time complexity of $O(\log n)$.

Average Search Time for RBT: 1299.2 ns

This average search time is much better than that of the BST, which is expected.

1.2.2.1. Code Implementation

I wrote an additional C++ function that demonstrates the randomized search operation in a BST and RBT. The function accepts the number of searches to perform and the filename of the CSV containing publisher names.

Explanation of Code

- **Loading Publisher Names:**
  - The function ‘returnNameVector(filename)‘ reads the publisher names from a CSV file and stores them in a vector of strings.
  - If the vector is empty, the function returns early, avoiding unnecessary processing.
- **Random Name Selection:**
  - A random number generator is created using the `random_device` and `mt19937` engine.
  - The range for random selection is set from 0 to the size of the publisher names vector minus 1.
  - This ensures a fair and uniform distribution of random indices.
- **Search Operation:**
  - A random publisher name is selected from the vector using the random index.
  - The search operation is executed via the `searchByName(random_search_key)` function.
- **Time Measurement:**
  - The time taken for each search operation is measured using the high-resolution clock from the `chrono` library.
  - The duration in nanoseconds is calculated and added to the cumulative search time.
- **Average Search Time:**
  - The search process is repeated `num_searches` times (50 in this experiment).

- The average search time is calculated by dividing the total search time by the number of searches.
- The result is displayed with a precision of one decimal place.

1.2.3. Comparison of Results

The results of the 50 random searches are summarized below, with the average search times for each tree structure.

Tree Type	Average Search Time (ns)
Binary Search Tree (BST)	1377.5
Red-Black Tree (RBT)	1299.2

Table 1.2: Comparison of Average Search Times for BST and RBT

1.2.4. Analysis of Results

The results showed that the Red-Black Tree performed slightly better, with searches completing on average 5.7% faster than the Binary Search Tree. This is because the RBT's balancing ensures logarithmic height regardless of the input order.

In contrast, the BST's performance depends on its structure. If the tree is unbalanced—such as when data is inserted in a sorted order—the height increases, causing slower searches. Even though the difference was modest in this test, it would become more noticeable with larger datasets.

Based on these results, the RBT is more consistent and reliable for searching, especially for datasets that may lead to unbalanced trees in a BST.

1.3. Best-Selling Publishers at the End of Each Decade

The result of the best-selling publishers at the end of each decade (1990, 2000, 2010, 2020) for both the Binary Search Tree (BST) and the Red-Black Tree (RBT) was both the same. However even though both trees use the same data, their internal **structure** is very different because of how they are built.

1.3.1. Best-Selling Publishers Results

The results for each marketplace (North America, Europe, and Rest of the World) are listed below. I had to print the results as i read them because i couldnt keep the year at the end of the reading process since the same publisher with different year meant adding them cumulatively so it was impossible to distinguish the decades after reading.

```
(base) feyzanur@Feyza-Dizustu-Bilgisayar HW3_BLG335E % ./solrbt VideoGames.csv
End of the 1990 Year
Best seller in North America: Nintendo - 160.02 million
Best seller in Europe: Nintendo - 30.03 million
Best seller rest of the World: Nintendo - 5.65 million
End of the 2000 Year
Best seller in North America: Nintendo - 334.75 million
Best seller in Europe: Nintendo - 101.97 million
Best seller rest of the World: Nintendo - 15.76 million
End of the 2010 Year
Best seller in North America: Nintendo - 722.26 million
Best seller in Europe: Nintendo - 350.91 million
Best seller rest of the World: Electronic Arts - 89.2 million
End of the 2020 Year
Best seller in North America: Nintendo - 814.43 million
Best seller in Europe: Nintendo - 418.36 million
Best seller rest of the World: Electronic Arts - 126.82 million
```

Figure 1.1: RBT and BST Solution Output (Same Result)

1.3.2. Pre-Order Traversal Outputs

Below are parts of the outputs generated by **pre-order traversal** of both BST and RBT. For the preorder printing i implemented a recursive function. I didnt use stack but that was also an option. I think the recursive one was much more simpler and easy to read compare to stack one i thought of.

```
----------Yamasa Entertainment
----------Xseed Games
----------Yacht Club Games
----------Yeti
-------Zoo Digital Publishing
--------Zenrin
--------Zushi Games
---------Zoo Games
---------fonfun
----------bitComposer Games
----------dramatic create
----------iWin
------responDESIGN
-------imageepoch Inc.
--------inXile Entertainment
(base) feyzanur@Feyza-Dizustu-Bilgisayar HW3_BLG335E %
```

Figure 1.2: BST Structure Output (Pre-Order)

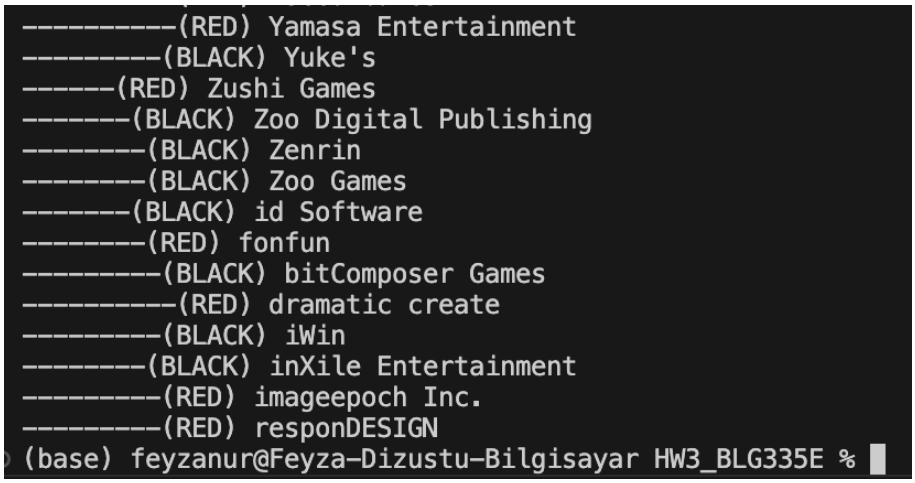

Figure 1.3: RBT Structure Output (Pre-Order with Colors)

1.3.3. What Does This Show?

From the results:

- Both BST and RBT correctly identify the best-selling publishers at each decade because they use the **same data**.
- The pre-order traversal shows the **same publishers** but in a slightly different order since how the tree is structured is different.

1.4. Final Tree Structure

1.4.1. Binary Search Tree (BST) Structure

The Binary Search Tree (BST) does not balance itself, so its structure can become unbalanced when data is inserted in sorted order. This results in a skewed shape, resembling a linked list, which increases the height of the tree.

- **Worst case**: If data is inserted in sorted order (ascending or descending), all nodes align to one side, causing the height of the tree to grow to $O(n)$, where n is the number of nodes.
- In a balanced BST, nodes are distributed evenly on both sides of the root, minimizing depth. However, in an unbalanced BST, one side of the tree becomes disproportionately deep while the other remains shallow or empty.
- This uneven structure makes the BST visually elongated, with long branches extending in one direction, making it structurally inefficient for storing and managing large datasets.

1.4.2. Red-Black Tree (RBT) Structure

The Red-Black Tree (RBT) is a self-balancing binary search tree. It maintains balance by using specific rules, such as assigning nodes a color (black or red) and performing rotations during insertions and deletions. These rules prevent the tree from becoming skewed, ensuring an even distribution of nodes.

- **Balancing mechanism**: The RBT maintains a balanced structure by limiting the height to $O(\log n)$, where n is the number of nodes. Even with sorted data, it redistributes nodes through rotations to prevent long branches on one side.
- **Structure**: The tree alternates black and red nodes, ensuring that no path from the root to a leaf is significantly longer than any other. This uniformity keeps the tree compact and symmetrical.
- **Height**: The maximum height of an RBT is logarithmic in the number of nodes, making it much shorter and more structured than an unbalanced BST.

Structural Comparison

RBT is balanced and looks even, no matter what order data is added. On the other hand, BST can become very uneven if data is not random.

Binary Search Tree (BST)	Red-Black Tree (RBT)
No balancing, tree can become skewed.	Self-balances with rules and rotations.
Can go up to $O(n)$ in worst cases.	Always $O(\log n)$ height.
Long, uneven branches.	Even, balanced, short branches.

Table 1.3: Comparison of BST and RBT Structures

In simple terms, RBT is more balanced and reliable, while BST can end up with long branches if the data isn't random.

1.5. Write Your Recommendation

I recommend Red-Black Tree (RBT) for managing this dataset with app. 16,000 rows. The reason is becuase RBT keeps itself balanced always. This means, even if the data is added in sorted order, tree height stays small ($O(\log n)$). Short height makes search, insert, and delete very fast.

Binary Search Tree (BST), on the other hand, has no balancing rule. If you add data like 1, 2, 3, ... tree becomes skewed, looking like a linked list. In this case, tree height can grow to $O(n)$, and all operations get slower.

RBT also fixes itself using rotations and node colors when needed. Even bad data ordering won't cause it to become unbalanced. For big datasets, this is very important to keep performance good.

In short, I pick RBT becuase it is better at handling large datasets and stays balanced all the time. BST may work for small datasets, but with 16,000 rows, it won't be efficient.

1.6. Ordered Input Comparison

First, I sorted the dataset by the *Name* column. After sorting, I used this ordered data to build both the BST and RBT. Because the data is ordered, each new node in the BST was added as the right child, creating a long, unbalanced structure. However, the RBT used rotations and balancing rules to fix itself.

Next, I ran 50 random searches on each tree. I recorded the time for each search in nanoseconds and calculated the average search time for both trees.

The results are as follows:

- **Red-Black Tree (RBT):**
  - **Insertion time:** 62,498 μs
  - **Average search time:** 2,641.6 ns
- **Binary Search Tree (BST):**
  - **Insertion time:** 14,202,389 μs
  - **Average search time:** 830,061.7 ns

1.6.1. What This Shows

- **Insertion Time:** The BST took much longer to insert data (14,202,389 μs) compared to the RBT (62,498 μs). This happened because the ordered data made the BST skewed, like a linked list, with all nodes on the right side. This makes the insertion time $O(n)$. On the other hand, the RBT stayed balanced by fixing itself during insertion, keeping its time at $O(\log n)$.
- **Search Time:** The BST search was very slow (830,061.7 ns), while the RBT search was much faster (2,641.6 ns). The unbalanced structure of the BST made searches take $O(n)$ time, but the balanced RBT handled searches in $O(\log n)$ time.

1.6.2. Conclusion

When using ordered data, the **Red-Black Tree (RBT)** is far better than the **Binary Search Tree (BST)**. The RBT stays balanced, so both inserting and searching are much faster and more consistent. The BST, without balancing, is not good for ordered input as it becomes slow and inefficient.