

Analysis of Algorithms

BLG 335E

Project 3 Report

RACHA BADREDDINE

badreddine21@itu.edu.tr

Faculty of Computer and Informatics Engineering

Department of Computer Engineering

Date of submission: 20/12/2024

1. Implementation

1.1. Data Insertion

To insert data into both the Binary Search Tree (BST) and Red-Black Tree (RBT), the CSV file was read line by line. The first two attributes, All attributes, the game name, platform, the year (used later to determine the best seller), publisher name, North America sales, Europe sales, and other sales were extracted. These values were stored in a string array, which was passed to the insertion function. the output of the insertion and runtime are shown in Figure 1.1 for BST and in Figure 1.2 for RBT.

1.1.1. Insertion Algorithm

The same algorithm was used for both structures since Red-Black Trees are a type of Binary Search Tree. The publisher name was used as the key to construct the tree in alphabetical order. The algorithm is as follows:

Algorithm 1 Insertion Algorithm for BST and RBT

Ensure: T with x inserted in the correct position

```
1: if  $T.root$  is null then
2:   Create the first node of the tree with key  $k$ 
3:   Set  $T.root \leftarrow x$ 
4: else
5:    $y \leftarrow T.root$ 
6:   while  $y \neq \text{null}$  do
7:     if  $k < y.key$  then
8:       if  $y.left \neq \text{null}$  then
9:          $y \leftarrow y.left$ 
10:      else
11:        Insert  $x$  as the left child of  $y$ 
12:        break
13:      end if
14:    else if  $k > y.key$  then
15:      if  $y.right \neq \text{null}$  then
16:         $y \leftarrow y.right$ 
17:      else
18:        Insert  $x$  as the right child of  $y$ 
19:        break
20:      end if
21:    else
22:      Update cumulative sales values
23:      break
24:    end if
25:  end while
26: end if
```

1.1.2. Balancing in Red-Black Tree

In addition to the above steps, the Red-Black Tree requires self-balancing. After inserting a node, a fix-up function is called to address potential violations of the Red-Black Tree properties. The three main cases handled during this process are:

1. **Case 1:** The uncle of the inserted node is red.
 - Recolor the parent and uncle to black, and the grandparent to red.
 - Move up the tree to check for further violations.
2. **Case 2:** The uncle of the inserted node is black.
 - **Sub-case 2.1:** The inserted node is a right child.
 - Perform a left rotation on the parent.
 - **Sub-case 2.2:** The inserted node is a left child.
 - Perform a right rotation on the grandparent.
 - Recolor the parent and grandparent appropriately.

These cases ensure the Red-Black Tree maintains its balanced structure, they were implemented and explained case by case in the code.

It can be seen that both structures took same runtime while being constructed, though RBT was faster than the BST and this will be explained in the following parts.

1.2. Search Efficiency

A total of 50 randomized searches were performed on both data structures. Since the data was random, no significant difference was observed since even Binary Search tree can be partially balanced with the random data, though Red-Black Trees were faster due to their balancing property that is always ensured. The search complexity in Red-Black Trees is always $O(\log n)$, while in Binary Search Trees, it is $O(\log n)$ on average but can be $O(n)$ in the worst case due to lack of balancing.

Here, for multiple execution of the code I got slightly different runtimes due to the randomized choice of publisher name. In BST the time sometimes was about 300ns and as explained before this is due to the lack of balancing but in average it was close to the Red Black tree.

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

The function to calculate the best seller at the end of each decade was called while inserting data into the structures. To optimize this task, we could have kept track of the best seller while reading the data using a public array, resulting in better performance. However, since the task required traversing the tree to find the best seller, an inorder traversal was used in both trees.

Inorder traversal follows the pattern Left \rightarrow Root \rightarrow Right. While traversing, the current best seller's sales and name were tracked until an update was required (when the sale is larger). This operation requires visiting all nodes in the tree, resulting in a complexity of $O(n)$ in both cases, regardless of the tree's shape or balancing. In addition to that, of course we will get the same results in both structures. the results are shown in Figure 1.1 for BST and in Figure 1.2 for RBT.

```
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
Time Taken to insert all data to BST: 125663µs
Average time for 50 random searches: 239.14 ns
```

Figure 1.1: Runtime of Binary Search Tree with unordered Data

```
root@d34f6d7998f6:/workspaces/code/src# ./rbt 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
Time Taken to insert all data to RBT: 82173µs
Average time for 50 random searches: 179.2 ns
```

Figure 1.2: Runtime of Red Black Tree with unordered Data

In this part, we were also asked to implement a preorder traversal function in the RBT, showing both the color and depth of the tree. I implemented an iterative preorder traversal following the rule **Root \rightarrow Left \rightarrow Right**.

I used two stacks: one for the nodes and the other for tracking the depth using dashes. Since the output is extensive, I wrote it not only to the terminal but also to a .txt file for easier inspection. A snapshot of the output is shown in Figure 1.3

```

(BLACK)Imagic
--(BLACK)Data Age
--(RED)BMG Interactive Entertainment
---(BLACK)Answer Software
----(BLACK)Activision
----- (RED)989 Studios
----- (BLACK)3DO
----- (RED)20th Century Fox Video Games
----- (BLACK)10TACLE Studios
----- (RED)1C Company
----- (BLACK)2D Boy
----- (RED)5pb
----- (BLACK)505 Games
----- (RED)49Games
----- (BLACK)989 Sports
----- (RED)7G//AMES
----- (BLACK)ASCII Entertainment
----- (BLACK)ASC Games
----- (RED)AQ Interactive
----- (RED)Acclaim Entertainment
----- (BLACK)ASK
----- (RED)ASCII Media Works
----- (RED)Abylight
----- (BLACK)Ackstudios
----- (RED)Accolade
----- (RED)Acquire
----- (RED)Agetec
----- (BLACK)Adeline Software
----- (BLACK)Activision Value
----- (RED)Activision Blizzard
----- (BLACK)Agatsuma Entertainment
----- (RED)Aerosoft
----- (BLACK)American Softworks
----- (RED)Alchemist
----- (BLACK)Aksys Games
----- (RED)Alawar Entertainment
----- (BLACK)Altron
----- (RED)Alternative Software
----- (RED)Alvion
----- (BLACK)Angel Studios
---(BLACK)Atari
---- (RED)ArtDink

```

Figure 1.3: Preorder output of RBT

1.4. Final Tree Structure

After implementing both the BST and RBT using random data, we observed that the BST was only partially balanced due to input randomization, while the RBT maintained a balanced structure as expected from its properties.

This difference is noticed especially from the average search runtime when searching for randomly generated keys. The BST was sometimes slower because its height could be larger than the balanced RBT, which maintains a height of $O(\log n)$ regardless of input order.

The maximum height of an RBTree is $2 \log_2(n + 1)$, here is the proof:

Let h_b = Black Height

Let h_{\max} = Maximum Height of Tree

Along a path:

$$N_{\text{black}} \geq \frac{h_{\max}}{2}$$

$$N_{\text{red}} \leq \frac{h_{\max}}{2}$$

Therefore:

$$\begin{aligned}N_{\text{path}} &\leq h_{\text{max}} + 1 \\h_{\text{max}} &\leq 2 \cdot h_b \\N_{\text{total}} &\leq 2^{h_{\text{max}}+1} - 1\end{aligned}$$

Combining inequalities:

$$2^{h_{\text{max}}+1} - 1 \leq N_{\text{total}} \leq 2 \cdot 2^{h_b} - 1$$

Solving for h_{max} :

$$h_{\text{max}} \leq 2 \cdot \log_2(n + 1) - 1$$

This guarantees that the RBT height is always $O(\log n)$, while the BST's height depends on the input data, making it only partially balanced. More details about this will be explained in Section 1.6.

1.5. Write Your Recommendation

After implementing both the RBT and BST and analyzing their outputs and runtimes, I recommend choosing the RBT for this task. The RBT maintains balance regardless of the input due to its properties and internal mechanisms, such as recoloring and rotations. This ensures consistent performance. While the BST is simpler to implement, it can degrade to linear search time in the worst case, as explained in the next section.

Here is a summary of the performance characteristics of both data structures:

Red-Black Tree (RBT)

- **Insertion:** $O(\log n)$ with additional $O(1)$ for rotations and color adjustments.
- **Fixing After Insert:** $O(1)$ to $O(\log n)$ depending on the number of rotations required.
- **Preorder:** $O(n)$ (traversing all nodes)

Binary Search Tree (BST)

- **Insertion:**
 - Average Case: $O(\log n)$
 - Worst Case: $O(n)$
- **Searching:**
 - Average Case: $O(\log n)$

- Worst Case: $O(n)$

In conclusion, the RBT is a better choice for this task.

1.6. Ordered Input Comparison

In this part, I implemented different cases (all commented in the code) since the instructions were ambiguous regarding what was expected.

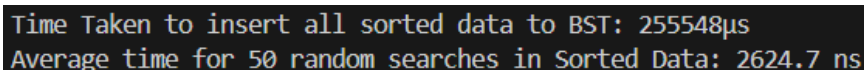
The goal of constructing the trees based on sorted data was to demonstrate that the BST is not always balanced. When given data in sorted order, the BST became a linked list. In contrast, the RBT maintains its balance property regardless of the input data order, resulting in better performance for all operations.

To reuse the functions implemented in previous parts, I read the data from files, stored it in a vector, sorted it based on the required attribute, and then wrote the modified data to another CSV file to construct the trees, as done earlier.

1.6.1. Sorting Based on Publisher Name

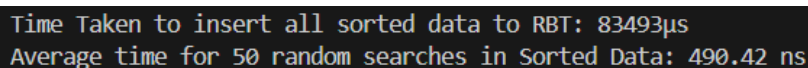
I began by sorting the data by publisher name, as this was the key used to construct the tree. After constructing the BST and RBT based on this data, the BST degenerated into a linked list, while the RBT remained a balanced binary tree, as expected. Consequently, the complexity of searching in the BST became $O(n)$, as previously mentioned, whereas the RBT maintained a complexity of $O(\log n)$.

The runtime for the BST is shown in Figure ??, and the runtime for the RBT is shown in Figure ?. As it can be noticed clearly, the RBT runtime was significantly faster than the BST.



```
Time Taken to insert all sorted data to BST: 255548µs
Average time for 50 random searches in Sorted Data: 2624.7 ns
```

Figure 1.4: BST based on sorted publisher name



```
Time Taken to insert all sorted data to RBT: 83493µs
Average time for 50 random searches in Sorted Data: 490.42 ns
```

Figure 1.5: RBT based on sorted publisher name

1.6.2. Sorting Based on Game Name

In the Document it was asked to sort by Name attribute which represents the Game name so i implemented this part too (code submitted needs some changed by uncommenting and commenting 2 lines). The problem here is that after sorting by Game name the publisher name was not in order or not even nearly sorted which resulted in similar performance to the previous parts where the BST was also partially balanced, RBT

always Balanced the result are shown in Figure 1.6 for BST, and the runtime for the RBT is shown in Figure 1.7. As it can be seen the performance of both structures is similar.

```
Time Taken to insert all sorted data to BST: 75026µs
Average time for 50 random searches in Sorted Data: 163.76 ns
```

Figure 1.6: BST storing Publishers based on sorted game name

```
##### Sorted Part #####
Time Taken to insert all sorted data to RBT: 73727µs
Average time for 50 random searches in Sorted Data: 204.2 ns
```

Figure 1.7: RBT storing Publishers based on sorted game name

Another case was implemented even though i do not think this is what we are asked to do, which is sorting and storing game name, i implemented this just to show the inefficiency of the BST when given sorted data by Game Name. Even though this violates our Node structure which holds the publisher object i just added it to compare the runtime. The results are in Figure ?? for BST, and the runtime for the RBT is shown in Figure 1.9. As it can be shown the BST was significantly slower than the RBT in both operations and this was due to the linked list structure. In addition to that this case was slower since we are holding game name which is unique for each line of data so the number of nodes was also significantly larger which led us to see these results.

```
Time Taken to insert all sorted data to BST: 943528µs
Average time for 50 random searches in Sorted Data: 48095 ns
```

Figure 1.8: BST based on sorted by game name

```
##### Sorted Part #####
Time Taken to insert all sorted data to RBT: 241647µs
Average time for 50 random searches in Sorted Data: 1163 ns
```

Figure 1.9: RBT based on sorted by game name