Analysis of Algorithms

BLG 335E

Project 3 Report

Kemal Tahir Bıcılıoğlu
bicilioglu21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 15/12/24

1. Implementation

1.1. Data Insertion

The main property of the binary search tree is that every node is larger than all the nodes in its left subtree and smaller than all the nodes in its right subtree. To preserve this property while inserting new data into the tree, we check the current node to see if it is greater or smaller than the node that should be inserted. If it is smaller, it means that we should go to the right subtree, and otherwise, the left subtree. We continue doing this comparison until we see a null node or a node with the same key, which means the node that should be inserted is already in the tree. This is $\mathcal{O}(\log n)$ time complexity if the tree is balanced since we always eliminate half of the nodes while searching for the place to insert. I accumulated the publisher sales when the node that should be inserted is already in the tree.

I measured the time taken to insert all the data and recorded the maximum depth of both trees; it can be seen in Figure 1 and Figure 2 :

	insertion	**max depth**
Binary Search Tree	48687ms	21
Red Black Tree	49210ms	11

Table 1.1: Comparison of Insertion Time and Tree Depth

The reason why the insertion times of the trees are similar, even though the Red-Black Tree has a smaller max depth, is that we are also calling the best sellers function when the decade passes. So, this affects the time measurement of the insertion. In the long run, the Red-Black Tree will insert new data into the tree in smaller time and also will search for a node in smaller time because it has a smaller depth than the binary search tree, meaning that it is more balanced.

1.2. Search Efficiency

As the name suggests, the search function for the binary search tree is very similar to the known array search function, binary search. We check the current node and continue to the subtrees accordingly. Because of that, the time complexity is $\mathcal{O}(\log n)$ if the tree is balanced. To compare the search time of the trees, I first take the publisher names into an array, then generate 50 random indices to search in the trees. After that, I call the search functions, measure the time, and take the average. Because the max depth of the Red-Black Tree is smaller, it is faster to search data in that tree.

I measured the time taken to search 50 random data and recorded the maximum depth of both trees; it can be seen in Figure 1 and Figure 2:

	searching	**max depth**
Binary Search Tree	1031ns	21
Red Black Tree	428ns	11

Table 1.2: Comparison of Search Time and Tree Depth

In fact, because the measurement is in nanoseconds, the gap is not that large since the max depth isn't that much different. However, the real distinction will be in the sorted data because the tree structures will vary more. The binary search tree won't balance itself, while the Red-Black Tree will.

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

I write a function named generate_list_of_best_selling which generates the best-selling publishers at the end of each decade. This basically continues to take data until a decade passes and inserts the data into the tree. When a new decade comes, it prints the best-selling publisher until that decade by calling the function find_best_seller, which also calls in_order_traversal. I wrote in_order_traversal to check all the nodes in the tree to find the best sellers for each category. I made the generate_list_of_best_selling function generic to years to show the best-selling publishers. The user can specify the years to see the best-selling publishers for whatever years they wish.

My in_order_traversal also finds the max depth of a tree so we can compare the growth of the max depths of the trees as my best selling function calls it. So we can see the depth when a decade passed.

max depths	**1990**	**2000**	**2010**	**2020**
Binary Search Tree	9	17	21	21
Red Black Tree	7	10	11	11

Table 1.3: Comparison of Tree Depth by Decades

Also, it can be seen in the Figure 3 and Figure 4.

We can see the balancing difference between the Red-Black Tree and the Binary Search Tree. As the data comes and is inserted into the trees, the Red-Black Tree efficiently grows by max depth, but the Binary Search Tree's max depth depends on the input order; it does not balance itself.

- I write a **preorder** traversal for red black tree which can be investigated from the figure Figure 5

1.4. Final Tree Structure

I tried to show the balancing difference between the Binary Search Tree and the Red-Black Tree with Figure 1 (shows the max depth of the BST), Figure 2 (shows the max

depth of the RBT), Figure 3 (shows the depth growth of the BST), and Figure 4 (shows the depth growth of the RBT).

The Red-Black Tree balances itself for each new data if its property is not satisfied, while the Binary Search Tree basically searches for the correct place to put the new data without balancing the tree.

1.5. Write Your Recommendation

While the Binary Search Tree is easy to implement and can achieve acceptable time for the operations —search, insert, and update data-, it fails when the data is ordered since it loses its balance, and the time complexity is no longer $\mathcal{O}(\log n)$ but $\mathcal{O}(n)$ as the tree becomes a linked list because of the ordered data insertions. The Red-Black Tree has functions to balance itself and checks whether the new data violates the coloring properties. If it does, it balances itself with rotations and coloring functions. Even though it is easy to make errors and hard to implement the Red-Black Tree, it is a very efficient data structure where we can achieve $\mathcal{O}(\log n)$ for all operations —search, insert, and update data— which is very efficient. To investigate different aspects of data, sometimes we may want to sort the data and insert them into the trees. In those cases, the Binary Search Tree should not be preferred, but the Red-Black Tree should be. However, if the data is not too big and expected to get small, large, and mid values randomly, it is better to use the Binary Search Tree since it is easy to debug and use for simpler cases.

1.6. Ordered Input Comparison

When we order the data by the publisher name which is key for the trees the Binary Search Tree becomes a linked list since all the insertions are in ascending order, and new values are always inserted to the right child of the leaf node. Because of that, the time complexity of the Binary Search Tree becomes $\mathcal{O}(n)$, which is not desired. On the other hand, even though the data is ordered, the Red-Black Tree always checks if the new data unbalances the tree and performs rotations and coloring to balance itself, achieving $\mathcal{O}(\log n)$ by guaranteeing the max depth of $2 \cdot \log_2(n+1)$ through its properties.

	searching	**max depth**
Binary Search Tree	5094ns	573
Red Black Tree	525ns	16

Table 1.4: Comparison of Insertion Time and Tree Depth in Ordered Data

- I measured the time taken to search 50 random data and recorded the maximum depth of both trees with the ordered data; it can be seen in Figure 1 and Figure 2.
- We can verify the fact binary search tree is like a linked list by looking its depth. There were 574 different publishers and the tree's max depth is 573.

- The difference between the max depth of the Red-Black Tree in the ordered data and the given data is balancing, and placing the data might vary depending on the order of the data for the Red-Black Tree. However, it always guarantees a max depth of $2 \cdot \log_2(n)$ for any order of the data by its properties.

1.7. Outputs

The image below shows the output produced by the program solution binary search tree when executed:

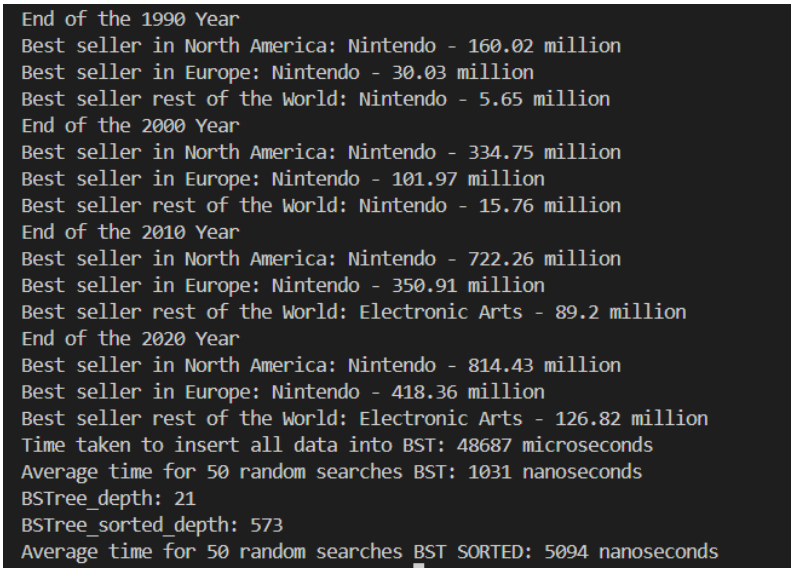

Figure 1.1: Binary Search Tree Output

The image below shows the output produced by the program solution red black tree when executed:

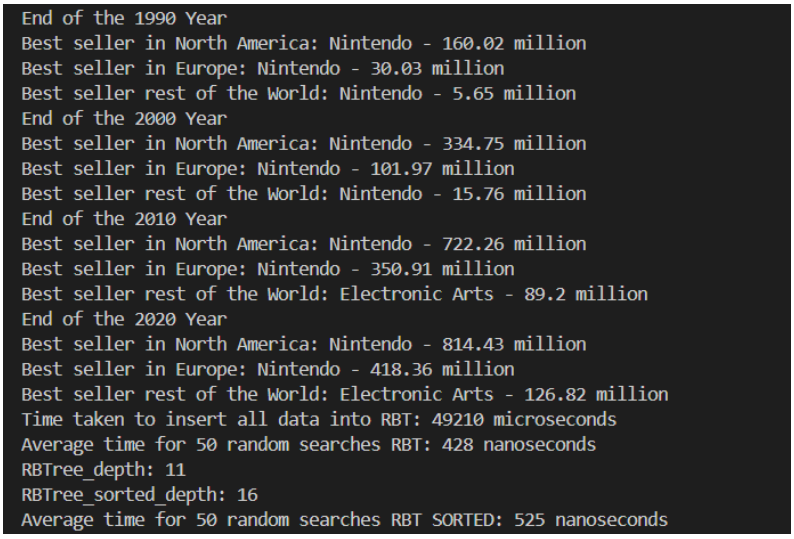

Figure 1.2: Red Black Tree Output

The image below shows the output produced by the program solution binary search tree with the max depth decade by decade:

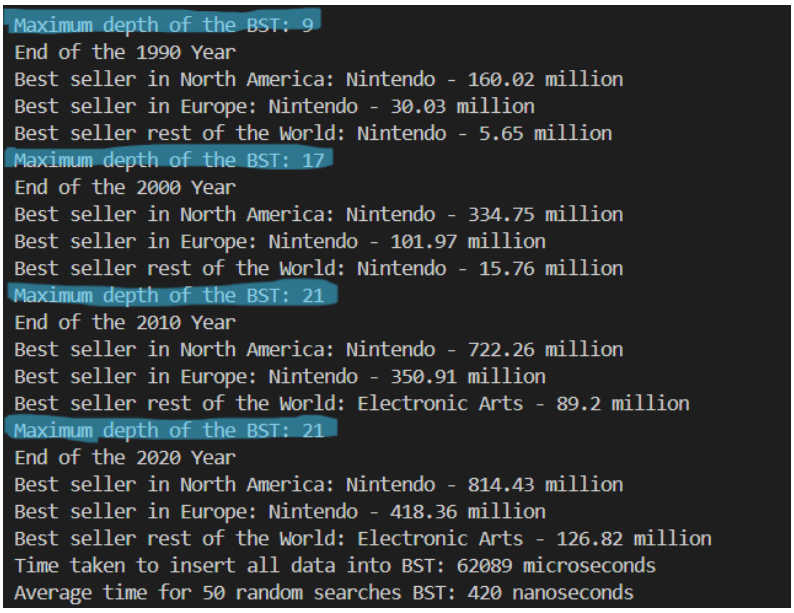

Figure 1.3: Binary Search Tree Depth Grow

The image below shows the output produced by the program solution red black tree with the max depth decade by decade:

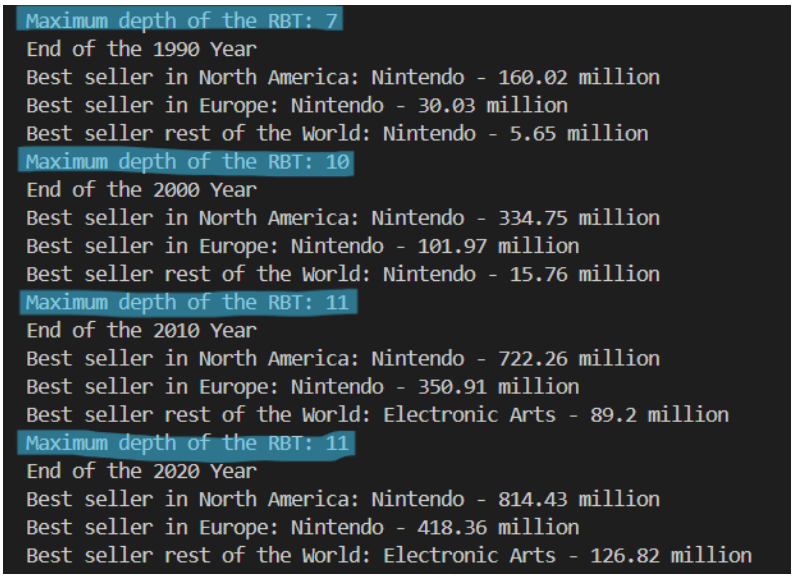

Figure 1.4: Red Black Tree Depth Grow

The image below shows the output produced by the program solution red black tree for the preorder traversal:

Figure 1.5: Red Black Tree Preorder