Analysis of Algorithms

BLG 335E

Project 3 Report

Alper Tutum
tutum21@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

The first task of the project was to implement insertion methods for both Red-Black Tree (RBT) and Binary Search Tree (BST). The algorithms work similarly but RBT's insert calls an additional $insertfixup()$ method in order to maintain the balanced structure of the tree. Durations of both insert functions are given below as well as the timing analysis.

	Duration (μs)
BS Tree	19135
RB Tree	19568

Table 1.1: Durations of insertion for different trees

Firstly, the time complexities of every function used in the insertion algorithms should be discussed:

- BST's insertion is fairly simple as it compares the key value of the input data to the root then recurses on the root's left or right child to reach the place where the data is to be inserted. This has an average time complexity of $O(logn)$. However if the keys are already sorted (or nearly sorted) every node will be appended to either the rightmost or leftmost side of the root. This is the worst case and it results in $O(n)$ time complexity. When the insertion of the entire dataset is taken into account the average case and worst case complexities become $O(nlogn)$ and $O(n^2)$ respectively.

- RBT's insertion works the same way as BST at first. However, after a new node is inserted into the tree a $insertfixup()$ subroutine needs to be called. Inside that subroutine are a fixed amount of rotation algorithms which are $O(1)$ time. Since the fixup can continue for the entire path from the newly inserted node to the root of the tree, it has a time complexity of $O(logn)$. RBT is a balanced binary search tree thus the height of the tree is bounded by $O(logn)$. This means the worst case for the insertion part is also $O(logn)$. This results in $O(logn)$ (for insertion) + $O(logn)$ (for fixup) = $O(logn)$ worst case time for the entire insertion algorithm. For inserting n nodes the worst case goes up to $O(nlogn)$.

The total time it takes to insert the entire dataset into the trees is almost the same. This is because with the given data BST performs in the average case $O(nlogn)$ and so does RBT. The reason RBT isn't more efficient is due to the fact that its insertion has more steps than BST. If the data was slightly sorted based on the nodes' chosen key, RBT would have performed visibly better than BST in this regard.

1.2. Search Efficiency

Second, a series of 50 random searches was to be performed on the two trees and the average time for the duration of one search was to be calculated. The search algorithm has the exact same implementation for both trees since RBT has the properties of a BST. A comparison is done in order to decide if the node being searched is to the right or the left of the current node. Once a node with the required key value is found, the search is complete. The time complexity analysis is as follows:

	Duration (ns)
BS Tree	704.96
RB Tree	602.4

Table 1.2: Durations of 50 random searches for different trees

The search algorithm is simply a traversal from the root node to the node with the value being searched for. This is the same for both BST and RBT thus the average time complexity of such an algorithm is $O(logn)$. Although, the worst cases differ for the two structures. As mentioned earlier, BST is not guaranteed to be balanced due to how the insertions are done thus leading to the problem of a tree with maximum height $O(n)$. This is not the case for RBT since the tree is balanced after each insertion to maintain the $O(logn)$ maximum height. This results in worst case time complexity of $O(n)$ for the BST's search subroutine and $O(logn)$ for RBT's.

If the duration times in Table 1.2 are compared, it can be seen that searching in RBT is slightly more efficient than in BST. If the data was arranged in the worst case for BST, the difference in efficiency would have been much higher.

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

The algorithm for finding the best selling publishers of a given tree of publishers utilizes pre-order traversal of every node in the tree. Once a node is visited, a comparison is done with the current best-seller of each region. If the current node has higher amount of sales, the best-seller for that region changes. This algorithm functions the same way in both BST and RBT data structures.

Since every node of a tree is visited exactly once, the time complexity of this algorithm is $O(n)$ for both types of trees.

The control for whether the end of the decade has been reached is done in the function which creates the tree from the given csv file. It checks the year attribute of every row and if the year surpasses the current decade, the best selling publishers are found and printed and the current decade is updated. This relies on the input data being sorted in ascending order of year of release.

1.4. Final Tree Structure

It can be concluded that the structure of the BST isn't the worst case with the given input data, although RBT is more efficient for new insertions and search operations. So BST is surely less balanced when compared to RBT.

1.5. Write Your Recommendation

With all of the above information, I recommend GameSoft to utilize the RBT to manage the video game sales data. As the size of the dataset increases with new publishers releasing new games, the $O(logn)$ worst case insertion of the RBT will outperform the BST implementation. Additionally RBT's search algorithm is guaranteed to run faster than BST's search, making it much more efficient for use cases where search operations are frequent.

1.6. Ordered Input Comparison

As mentioned earlier if the input data is in sorted form with respect to the key for insertion, BST is expected to perform its insertion and search operations in $O(n)$ time. Making insertion of the entire dataset work in $O(n^2)$. The results for the timing measurements in that case is given below:

	Insertion (μs)	Search (ns)
BS Tree	328249	14966.7
RB Tree	14925	673.36

Table 1.3: Durations of functions with ordered input

The measured duration times in Table 1.3 supports the expectations. Both functions of RBT perform in similar time, maintaining the $O(nlogn)$ and $O(logn)$ time complexities due to the balanced nature of the tree. The same cannot be said for BST's case. The transition from Tables 1.1 and 1.2 to Table 1.3 can only be explained by the change in time complexity from $O(nlogn)$ to $O(n^2)$ for insert and $O(logn)$ to $O(n)$ for search. So ordering the input changed the time complexity of BST's methods to match the worst case whereas RBT's methods kept their relatively low time complexity.