Analysis of Algorithms

BLG 335E

Project 3 Report

EMRE YAZICI
150220063@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

Initially, rows were read from the CSV file. Each row's tokens were then sent to the `insertValue` function as a `vector<string>`. Within the `insertValue` function, the tokens were destructured and passed to the `insert` function. In the `insert` function, if a publisher with the same name already existed in the tree, its `sales` value was updated; otherwise, a new node was added. During this process, the search operation's complexity was $\Omega(\log n)$ in the best case and $O(n)$ in the worst case for the BST, while it was consistently $O(\log n)$ for the RBT. Throughout this process, every time the `insert` function was called, the values in the `best_seller` array were also checked and updated if necessary, thus saving time for finding the best seller.

While the above process was the same for both BST and RBT, in RBT, the `insertFixup` function was called each time a new publisher was added to ensure the tree remained balanced. Since the `insertFixup` function traverses from the newly added node to the root, it incurs a cost of $O(\log n)$.

In the analysis performed, the total time taken to construct the tree was 8785.2 microseconds for the BST, while it was 8134.29 microseconds for the RBT.

1.2. Search Efficiency

After constructing the trees, search operations were performed for 50 random `publisherName`s. According to the analysis, the average search time for the BST was 501 nanosecons, while it was 424 nanoseconds for the RBT. These results demonstrate the advantage of a balanced tree. While the time complexity for the RBT is $\Theta(\log n)$, the BST encountered a time complexity of $\Omega(\log n)$ in the best case and $O(n)$ in the worst case.

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

We ensured decade control while reading rows from the CSV file. When we detected the end of a decade, we called the `print_best_sellers` function within the `generate_BST_tree_from_csv` function. Finding the best sellers required no additional computational overhead since the `best_seller` array was already being updated during the `insert` operations. This allowed us to perform this operation in $O(1)$ time. In the analysis, the average time for this operation was as low as 19 nanoseconds for both BST and RBT.

1.4. Final Tree Structure

In our example dataset, the input order was close to optimal, resulting in a BST that was nearly balanced, as observed in our search efficiency analysis. However, the input order may not always be this optimal. In the worst-case scenario, such as with sorted input, the tree would degenerate into a linked list. This situation results in a complexity of $O(n)$ in the worst case and $\Omega(n)$ in the best case for the BST.

1.5. Write Your Recommendation

Since the likelihood of having optimally ordered data is generally low, using a balanced tree is more advantageous. The main drawback of balanced trees is the extra fix-up operations required during insert and delete operations. However, they compensate for this during the search phase of insert and delete operations, making them more efficient overall.

1.6. Ordered Input Comparison

In this section, we sorted the publishers by `publisherName` and constructed a tree accordingly. We then performed 50 random searches and analyzed the results. The average insert time for the BST was measured as 6578 nanoseconds, whereas it was 620 nanoseconds for the RBT. This difference is due to the search phase during each insert operation being $O(n)$ for the BST. For the search operation, the average time was measured as 5377 nanoseconds for the BST and 475 nanoseconds for the RBT.