Implementation of Red and Black Tree

A decision was made to recreate the Node structure and the RBTree structure from “scratch. So, two different classes were made to support the implementation. The respective classes are the RBTree class and the Node class.

RBTree and Node Classes

In the given classes there exist instance variables that help the insertion, deletion, and procedures operate. For the node class there are instance variables of .left and. right to be able to access their children, and their children can access their own children with the. left and. right etc. (achieving the recursive property of a Binary Search Tree). There are other instance variables that are helpful for other implementations. For example, the. isRoot and .is_right_child come very handy when handling insertions and rotations.

The RBTree has instance variables of its own. The most important are the root, and the size, this helps keep track of what the root is and what size the RBTree is at.

RBTree Methods

To handle tree insertion, the RBTree class has a method called insertTree that takes advantage of the binary tree structure. This class iterates through the tree and depending on if the node value is less than, or greater than the specified node value, it will either go to the left child or the right child. This process iterates until the node value is None which then it will append that node value to the left child (if less than) or right child (if greater than). After the node is inserted, a method called checkRotations is called to make sure the RBTree properties are satisfied. If they aren’t satisfied, then the checkRotations method handles the recoloring and the rotations of every case specified in RBTree class slides.

To expand on the checkRotations method, checkRotations handles the three cases (plus their respective mirrored scenarios). These cases account for the different arrangement and colors that the RBTree may contain. This method guarantees that the RBTree properties stay intact.

As per the requirements the method deleteFromTree was also added, it takes in a node value as a parameter and removes it from the tree. This method iterates through the RBTree structure similarly to the tree insertion method, except for when it finds the given node with the corresponding value and removes it from the tree. This part of the project was the most difficult, as the pointer values for the removed node had to removed is a specific order. This

method also uses a helper method defined as findLargestFromLeftSubtree(node). This method finds the right most node from the given node's left subtree. It is utilized when a node is removed and it has both a right subtree and left subtree. The overall purpose of this method is to use the returned node to replace the deleted node to retain the Binary Search Tree properties.

Also, similarly to the insertInTree method, the deleteFromTree has its own checkRotations method called checkRotationsForDeletion. This method has the same purpose as the checkRotations method; however, it is used to maintain the RBTree properties post deletion.

Per the project requirements the class also includes a search method. This method's name is searchTree(value), it takes any given value and iterates through the RBTree with the same methodology as the insert and delete method. The method will return True if the value exists in the tree and false if it doesn't.

Also to fulfill the project requirements a traversal method was included. This method is InOrderTraversal and outputs an array in the in-order traversal on the binary search tree. An in-order traversal was chosen over the other two traversal methods because of its wide use and familiarity within the Computer Science discipline.

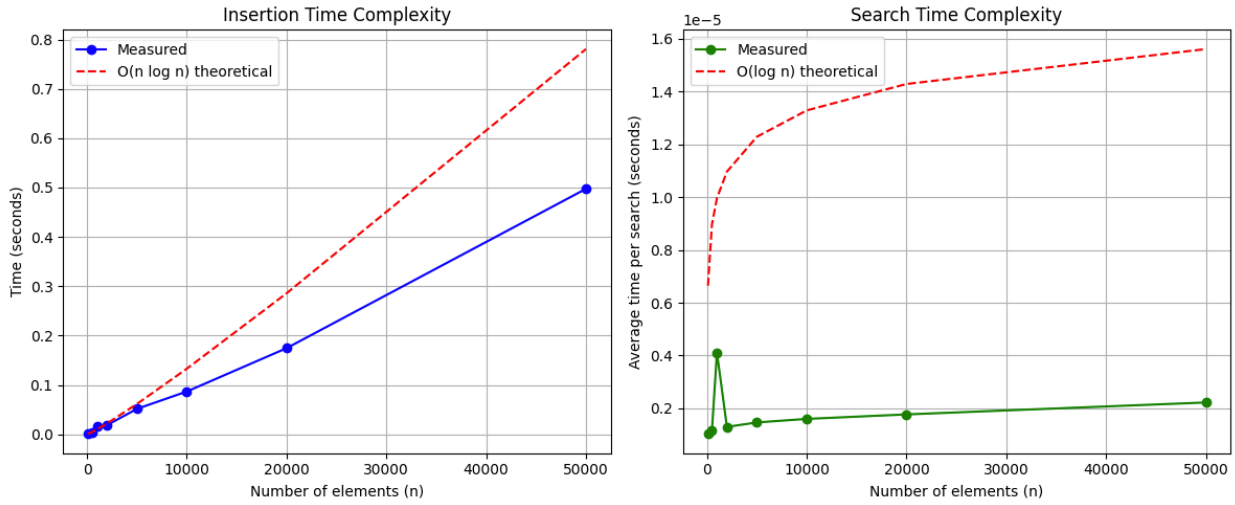
Testing

For testing, another python script was delegated to test the methods of the RBTree. The "testing_suite.py" contains tons of test case scenarios to test the methods of the RBTree.

Performance analysis report

Time Complexity of RBTree

Much like the testing, another python script was delegated to see the performance of the RBTree. This python script utilized NumPy and matplotlib to create a visualize large sets of data to accurately represent “n”. The “time_complexity_suite.py” creates visualization for the measure amount of time (in seconds) that it takes to run the insertion and the search time. As seen in the figure below,



Although it may seem that the time is not accurate to the theoretical values. The overall shape is comparable to each theoretical graph. It is also safe to say as the graph is scaled to a larger n, the measure values will depict the theoretical values more accurately. What matters the most in these graphs is the shape of the graph rather than the sizes of the output.

The ideal time complexity for the insertion time is log(n). The reason the graph is displaying n log(n) is because it is accounting for the time of inserting all the items up to n, mathematically making it log(n) for each individual insertion.

Comparisons with other Data Structures

Data Structure	Search	Insert	Delete
Red-Black Tree	O(log n)	O(log n)	O(log n)
AVL Tree	O(log n)	O(log n)	O(log n)
Hash Table	O(1) avg, O(n) worst	O(1) avg, O(n) worst	O(1) avg, O(n) worst

Red-Black Tree compared to AVL Tree

Both are self-balancing binary search trees with O(log n) operations, yet they vary in the strictness of their balance. AVL trees ensure a tighter balance (height difference ≤ 1 between subtrees), which leads to quicker searches but slower insertions and deletions. Red-Black trees utilize color attributes for looser balancing, resulting in quicker modifications.

AVL trees are highly effective in read-intensive scenarios like databases that require frequent searches and static datasets with rare modifications. Red-Black trees are favored for applications with heavy write operations and are commonly utilized in standard libraries (C++ std::map, Java TreeMap) and the scheduler of the Linux kernel because of their consistent performance in all operations.

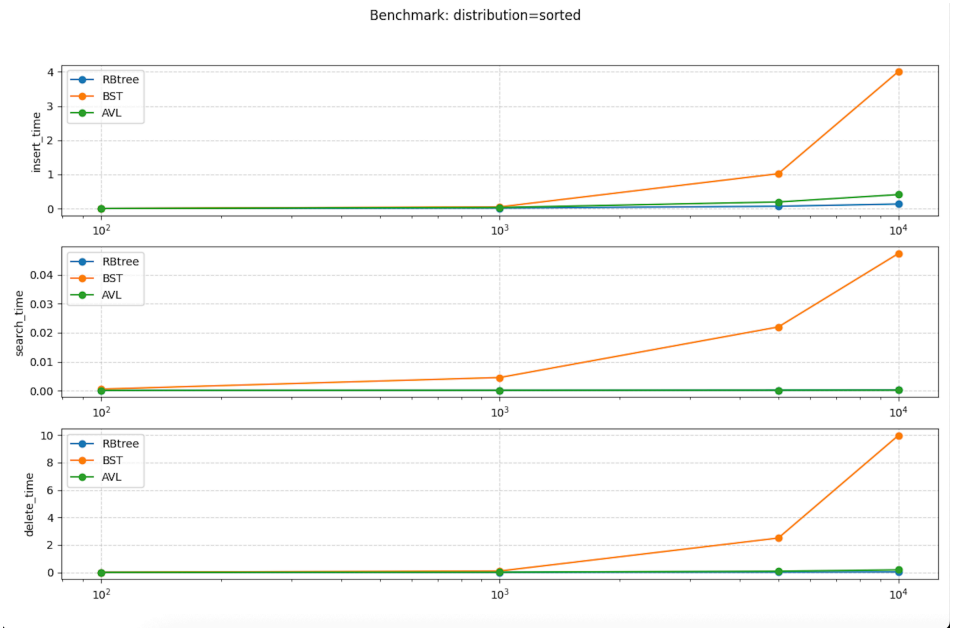
Red-Black Tree compared to Hash Table

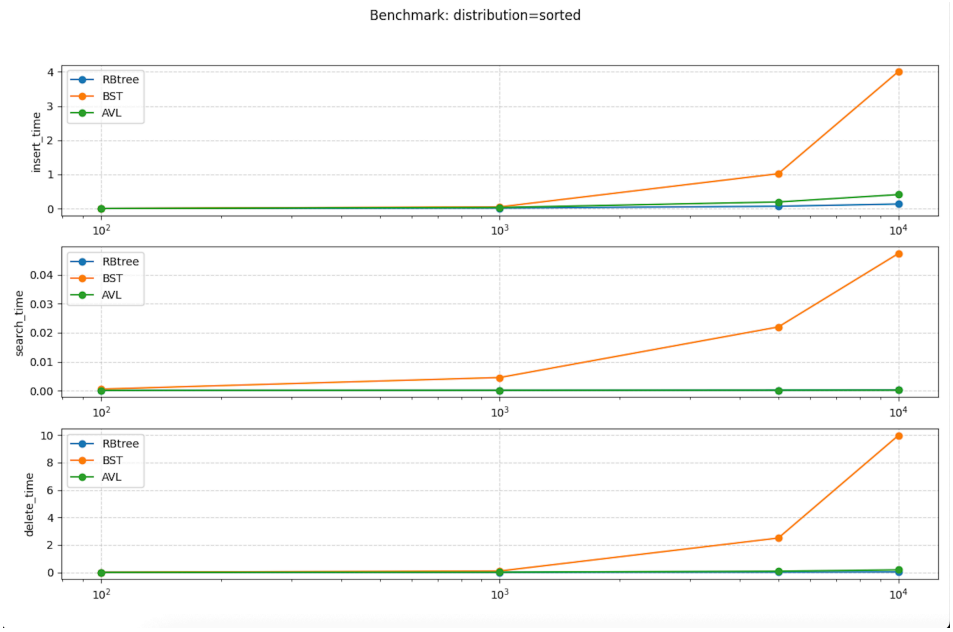
Hash tables ensure O(1) average-case performance but do not support ordering, whereas Red-Black trees offer O(log n) operations along with complete ordering functionality. Hash tables provide superior average-case speed and easier implementation for fundamental key-value storage.

Red-Black trees ensure a worst-case performance of O(log n), while hash tables can deteriorate to O(n) when facing collision chains. In situations where ordering is important, Red-Black trees enable efficient ordered iteration and range queries in O(k + log n) time, whereas hash tables operate at O(n). They do not need a hash function and operate with any comparable data type. Red-Black trees enable effective min/max operations in O(log n) compared to O(n) for hash tables, offering improved memory efficiency by eliminating load factor overhead.

Hash tables are perfect for unordered searches when average performance is a priority. Red-Black trees excel in dynamic ordered datasets and are chosen by industry for their balanced performance across all operations.

Another performance analysis allowed the comparison of different types of inputs. In this file it outputted different plots comparing the time complexities for three different binary search trees for different types of inputs.





Link to the Github repository →

https://github.com/Arrnimon/FoundationsFinalProject