Binary Search and Red-Black Trees

1. Abstract:

In this study, we implement a new Binary Search Tree by building upon the existing Red-Black Tree implementation. The focus lies on updating the insertion routine to handle duplicate values effectively. When confronted with duplicate values during insertion, the system avoids adding them to maintain uniqueness. Additionally, modifications are made to the Inorder Tree Walk algorithm for both Binary Search Tree and Red-Black Tree. This adjustment ensures the traversal of the tree and the orderly copying of its elements back to an array, excluding duplicates. The final step involves returning the count of elements copied into the array. The study further involves tracking and recording various occurrences during sequences of insertions, including the count of duplicates, insertion cases exclusive to Red-Black Trees, and the count of left and right rotations specific to Red-Black Trees. Lastly, the study counts and returns the number of accessible Black nodes along the path. The runtime performance is measured across different input sizes, and a comprehensive report is provided, analyzing and concluding on the observed running time behavior.

2. Result:

Two different sorting algorithms are implemented and the time complexity of each one of them was measured to complete the given experiment. The two algorithms are Binary Search Tree, Red-Black Tree.

- **Binary Search Tree (BST):**

A Binary Search Tree, a type of node-based binary tree data structure, adheres to the following principles:

→ The left subtree of any node exclusively contains nodes with keys less than the node's own key.
→ Conversely, the right subtree of a node solely consists of nodes with keys greater than the node's key.
→ Both the left and right subtrees must individually qualify as binary search trees.

Each node in the tree possesses a key and a corresponding value. During a search operation, the target key is compared with the keys in the BST. If a match is found, the associated value is returned. The search process initiates from the root node, and if the desired data is less than the current node's key, the search continues in the left subtree. On the other hand, if the data is greater, the search proceeds in the right subtree.

Algorithm for Insertion:
Step 1: Declare the nodes bs_tree_node* x and bs_tree_node* y, initializing them as y = T_nil and x = T_root.
Step 2: Utilize a while loop (while (x != T_nil)) to check for duplicate values during insertion.
Step 3: Compare the value to be inserted with the root node. If it is less, move to the left; otherwise, go to the right.
Step 4: Traverse the tree until finding the ideal location for insertion or reaching the last leaf node.

Binary Search Tree Inorder Traversal:
In a Binary Search Tree, Inorder traversal is a method for navigating the tree. This algorithm explores the left subtree initially, followed by the root, and eventually the nodes on the right subtree. The traversal begins at the root, moves to the left node, iteratively explores further left nodes until a leaf node is reached. Subsequently, the value of that node is copied into an array, and the process continues until all nodes have been visited.
Algorithm for Inorder Traversal:
Step 1: Initialize the traversal from the root and move to the left node until a leaf node is reached.
Step 2: Increment the array index counter to copy the node value into the array.
Step 3: Visit the root node.
Step 4: Traverse the right subtree by calling inorder_output(x->right, level+1, array).

- **Red-Black Tree**

The Red-Black Tree, also known as a self-balancing Binary Search Tree, includes an additional field for each node, apart from the key and value, which is referred to as "color." The tree uses only two colors, Red and Black, to maintain

balance during insertion and deletion operations. This coloring mechanism enhances the efficiency of the Red-Black Tree compared to a Binary Search Tree by reducing search times.
The key properties of the Red-Black Tree include:

→ Each node is assigned a color, either red or black.
→ The root node is always black.
→ No two adjacent nodes can be red, meaning a red node cannot have a red parent or red child.
→ Every path from a node, including the root node, to any of its descendant NULL nodes contains the same number of black nodes.

<u>Algorithm for Insertion:</u>
Step 1: Begin by declaring two nodes, bs_tree_node* x and bs_tree_node* y, and initialize them as follows: y = T_nil, x = T_root.
Step 2: Use a while loop to iterate as long as x is not equal to T_nil, checking for the presence of any duplicate values to be inserted.
Step 3: Compare the value to be inserted with the root node, taking into account the color of the node in the Red-Black Tree. Move to the left if the value is less, otherwise, go to the right.
Step 4: Ensure that no two consecutive red nodes are present.
Step 5: Traverse the tree until finding the optimal location for inserting the value or reaching the last leaf node.

<u>Red-Black Tree Inorder Traversal:</u>
In the Red-Black Tree, the Inorder traversal is a technique for navigating the tree. This algorithm prioritizes exploring the left subtree, followed by the root, and finally, the nodes in the right subtree. Start the traversal from the root, move to the left nodes, and continue this process until reaching a leaf node. After reaching a leaf, copy its value into an array, and repeat this procedure until all nodes are visited.
<u>Algorithm for Inorder Traversal:</u>
Step 1: Traverse the left subtree by calling inorder_output(x->left, level+1, array).
Step 2: Increment the array index counter to copy the node value into the array.
Step 3: Visit the root node.
Step 4: Traverse the right subtree by calling inorder_output(x->right, level+1, array).

<u>Testing Values:</u>
1. <u>Binary Search Tree and Red-Black Tree:</u>

- The runtime in ms for Binary Search Tree and Red-Black Tree are as follows:

Size (n)	Descending Order, D = -1		Random Order, D = 0		Ascending Order, D = 1	
	BST (ms)	RBT (ms)	BST (ms)	RBT (ms)	BST (ms)	RBT (ms)
5000	48	1	0	1	49	2
10000	172	3	3	3	176	2
25000	1029	7	8	35	1059	7
50000	4082	13	19	21	4254	15
100000	18063	32	42	51	18668	31
250000	-	86	135	134	-	86
500000	-	166	360	407	-	169

- When the given array is in descending order (D = -1), Binary Search Tree exhibits significantly greater runtime compared to Red-Black Tree.
- The runtime of Binary Search Tree escalates significantly with an increase in the size of the array.
- Red-Black Tree, for an array of the same size, demonstrates remarkably shorter sorting time, providing the expected outcome efficiently.

- Descending order poses a worst-case scenario for Binary Search Tree, making the Inorder Tree Walk operation challenging.
- The process of visiting each node and retrieving its value becomes time-consuming.
- **Figure 1** presents a comparison of the time taken by Binary Search Tree and Red-Black Tree when the array is in reverse order (D = -1).
- From **Figure 1**, we can say that Red-Black Tree takes very less time even though the input size increases as compared to Binary Search Tree.
- **Figure 2** represents input size vs Runtime for the Random order.
- **Figure 2** shows that in case of random sorted arrays, Red-Black Tree roughly takes same amount of time but as the input size increases, we can see a leap in the time taken by the Red-Black Tree.
- Binary Search Tree is more efficient when it comes to Random sorted arrays.
- **Figure 3** represents input size vs Runtime for the Ascending order.
- From **Figure 3**, we can say that in case of Sorted order array, Binary Search Tree requires a lot of time to traverse through the data. It can also be considered as a Worst-case scenario similar to the descending order case.
- The Red-Black Tree takes very less time and is very efficient even though if the input size increases, time taken is very less as compared to Binary Search Tree.
- For Binary Search Tree, as the input size increases, the time taken by the algorithm also increases

Descending Order, D = -1

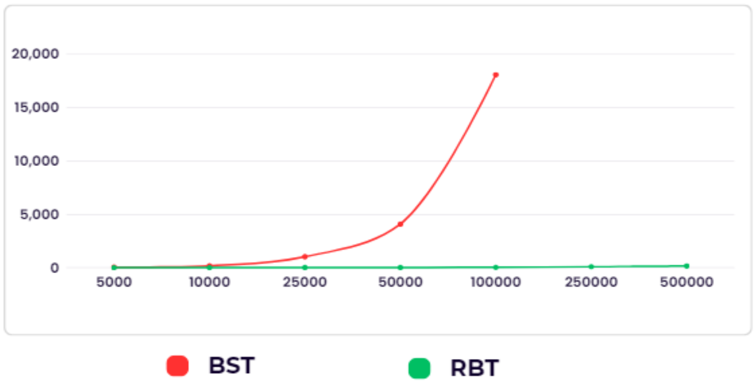


Figure 1

Random Order, D = 0

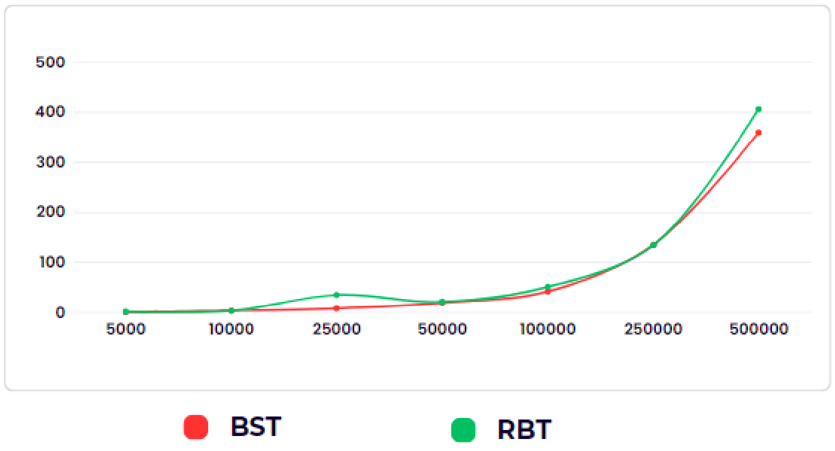
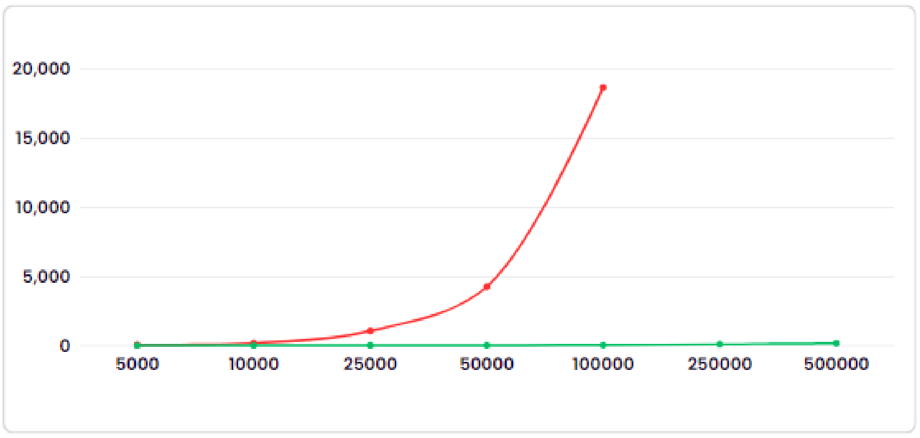


Figure 2

Ascending Order, D = 1



🟥 BST 🟩 RBT

Figure 3

2. **Average Values of Counters and Duplicates:**

- The below tables show readings of the average values of the counters for all the combinations of input size and direction.
- Since the array is reversed sorted, there are 0 instances for case 2 and left rotation for Red-Black Tree.
- Similarly, when the array is sorted, there are 0 instances for case 2 and right rotation for Red-Black Tree.

Size (n)	Descending Order, D = -1						
	Counter Duplicates BST	Counter Duplicates RBT	Counter Case 1	Counter Case 2	Counter Case 3	Left Rotate	Right Rotate
5000	0	0	4973	0	4978	0	4978
10000	0	0	9971	0	9976	0	9976
25000	0	0	24968	0	24973	0	24973
50000	0	0	49966	0	49971	0	49971
100000	0	0	99964	0	99969	0	99969
250000	0	0	249961	0	249967	0	249967
500000	0	0	499959	0	499965	0	499965

Size (n)	Descending Order, D = 0						
	Counter Duplicates BST	Counter Duplicates RBT	Counter Case 1	Counter Case 2	Counter Case 3	Left Rotate	Right Rotate
5000	0	0	2553	945	1948	1002	946
10000	0	0	5106	1884	3847	1924	1923
25000	0	0	12821	4822	9724	4920	4804
50000	0	0	25624	9801	19542	9792	9750
100000	2	1	51267	19416	38893	19469	19424
250000	15	12	128427	48884	97138	48693	48445
500000	54	58	256400	97146	194210	97097	97113

Size (n)	Descending Order, D = 1						
	Counter Duplicates BST	Counter Duplicates RBT	Counter Case 1	Counter Case 2	Counter Case 3	Left Rotate	Right Rotate
5000	0	0	4973	0	4978	4978	0
10000	0	0	9971	0	9976	9976	0
25000	0	0	24968	0	24973	24973	0
50000	0	0	49966	0	49971	49971	0
100000	0	0	99964	0	99969	99969	0
250000	0	0	249961	0	249967	249967	0
500000	0	0	499959	0	499965	499965	0

3. Discussion:

a) Running Time:

• The Binary Search Tree maintains balance, resulting in a logarithmic height (log(n)), leading to an overall time complexity of O(logn) for Binary Search Tree operations.

• With each iteration or recursive call, the search is halved, contributing to the efficiency of Binary Search Tree operations.

• The time required for Binary Search Tree operations is notably high in cases of ascending or descending order, representing a worst-case scenario.

• In scenarios with a random order, Binary Search Tree demonstrates efficiency, requiring less time.

• The Red-Black tree boasts an average time complexity of O(1), with the worst-case complexity being O(logn).

• Red-Black trees exhibit efficiency in insert and delete operations due to their minimal structural changes to maintain balance.

• Often referred to as a self-balancing binary search tree, the Red-Black tree outperforms the binary search tree in terms of time efficiency, showcasing significantly faster performance.

b) Limitations:

• The recursive nature of the Binary Search Tree demands a significant amount of stack space.

• The Binary Search Tree algorithm is acknowledged for its error-prone and intricate nature.

• Binary Search Trees exhibit poor caching capabilities due to their support for random data access.

• The execution of Red-Black Trees is recognized for its complexity.

c) Improvements:

• Enhancements can be made to the Binary Search Tree implementation when dealing with data in ascending order. One approach is transforming the Binary Search Tree into a height-balanced binary tree or a self-balancing binary tree. This modification is aimed at enhancing the traversal time within the updated Binary Search Tree.

• Another strategy to enhance the running time of the binary search tree involves implementing node caching. Storing and retrieving nodes through caching can contribute to improved efficiency in the operation of the binary search tree.

4. Conclusion:

Based on the aforementioned results, it can be asserted that the Binary Search Tree exhibits slightly better efficiency than Red-Black Trees in scenarios involving random order. However, in cases of ascending and descending order, the Binary Search Tree experiences significantly higher execution times compared to Red-Black Trees, categorizing it as a worst-case scenario. As the input size escalates, the sorting time for Binary Search Trees increases rapidly. Additionally, it is notable that the likelihood of encountering duplicate values is higher in random arrays compared to arrays in ascending or descending order.