CS590 homework 3 – Binary-Search, and Red-black Trees

1. Abstract:

We have the implementation of the Red-Black Tree and with the help of that, we need to implement a new Binary Search Tree. We must update the insertion routine and at the same time handle the duplicate values. If the duplicate value arrives, then we must not insert that duplicate value. We also need to modify the Inorder Tree Walk algorithm for Binary Search Tree and Red-Black Tree in a way such that it will traverse the tree and copy its elements back to the array in sorted order. Duplicates should not be copied as they are already eliminated. In the end, we need to return number of elements which are copied into the array. Furthermore, we need to count the following occurrences over the sequences of insertions.

- Counter for number of duplicates.
- Counter for each of the insertion cases for Red-Black Tree only.
- Counter for left rotate and for right rotate for Red-Black Tree only.

Lastly, we need to count and return the number of Black nodes which are accessible on the path. For all the cases, we need to measure the runtime performance for different input sizes and provide a report and conclusion on how the running time behaves.

2. Result:

Testing sorting algorithms

Two different types of sorting algorithm were implemented and the time complexity of each one of them was measured to complete the given experiment. The two algorithms are as follows:

- Binary Search Tree
- Red-Black Tree

Binary Search Tree:

Binary Seach Tree is a node based binary tree data structure and it contains the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is returned. The searching always begins from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree

General Algorithm for Insertion:

- Step 1: Declare the following nodes which are bs_tree_node* x, bs_tree_node* y and initialize them to y = T_nil, x = T_root;
- Step 2: Declare a while (x != T_nil) to check whether there are any duplicate values which are to be inserted.
- Step 3: Now we need to check whether the value to be inserted is greater or lesser than the root node. If the value is less then we go to the left, otherwise we go to the right.
- Step 4: We traverse the tree till me find the ideal location for the value to be inserted or till the last leaf node.

Binary Search Tree Inorder Traversal:

In the Binary Search Tree, Inorder traversal is a way to traverse the tree. During the in-order traversal algorithm, the left subtree is explored first, followed by root, and finally nodes on the right subtree. You start traversal from root then goes to the left node, then again goes to the left node until you reach a leaf node. After that, we copy that value into the array and continue the process till all the nodes are visited.

General Algorithm for Inorder Traversal:

- Step 1: Traverse the left sub tree, i.e., call inorder_output(x->left,level+1, array);
- Step 2: Increment the counter index of the array to copy the node value into the array.
- Step 2: Visit the Root node.
- Step 3: Traverse the right sub tree, i.e., call inorder_output(x->right,level+1, array);

Red-Black Tree:

Red-Black Tree is also called as self-balancing Binary Search Tree. Each node along with key and value, stores an extra field which is color. There are only two colors which are stored in this tree i.e., Red and Black. These colors are used to ensure that the tree remains balanced during insertions and deletions operations. These property helps in reduce the searching time and makes it more efficient as compared to the Binary Search Tree. The properties of the Red-Black Tree are as follows:

- Every node has a color either red or black.
- The root of the tree is always black.
- There are no two adjacent red nodes i.e., A red node cannot have a red parent or red child.
- Every path from a node including the root node to any of its descendants' NULL nodes has the same number of black nodes.

General Algorithm for Insertion:

- Step 1: Declare the following nodes which are bs_tree_node* x, bs_tree_node* y and initialize them to y = T_nil, x = T_root;
- Step 2: Declare a while (x != T_nil) to check whether there are any duplicate values which are to be inserted.
- Step 3: Now we need to check whether the value to be inserted is greater or lesser than the root node. We also need to check the color of the node which is present in the Red-Black Tree. If the value is less then we go to the left, otherwise we go to the right.
- Step 4: We must make sure that no two red nodes are together.
- Step 5: We traverse the tree till me find the ideal location for the value to be inserted or till the last leaf node.

Red-Black Tree Inorder Traversal:

In the Red-Black Tree, Inorder traversal is a way to traverse the tree. During the in-order traversal algorithm, the left subtree is explored first, followed by root, and finally nodes on the right subtree. You start traversal from root then goes to the left node, then again goes to the left node until you reach a leaf node. After that, we copy that value into the array and continue the process till all the nodes are visited.

General Algorithm for Inorder Traversal:

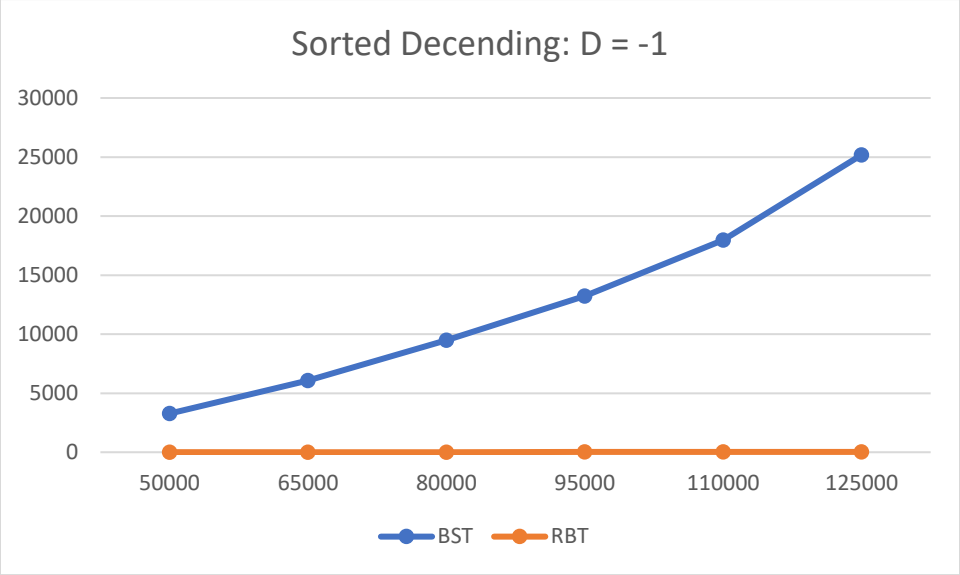
- Step 1: Traverse the left sub tree, i.e., call inorder_output(x->left,level+1, array);
- Step 2: Increment the counter index of the array to copy the node value into the array.
- Step 2: Visit the Root node.
- Step 3: Traverse the right sub tree, i.e., call inorder_output(x->right,level+1, array);

Testing Values:

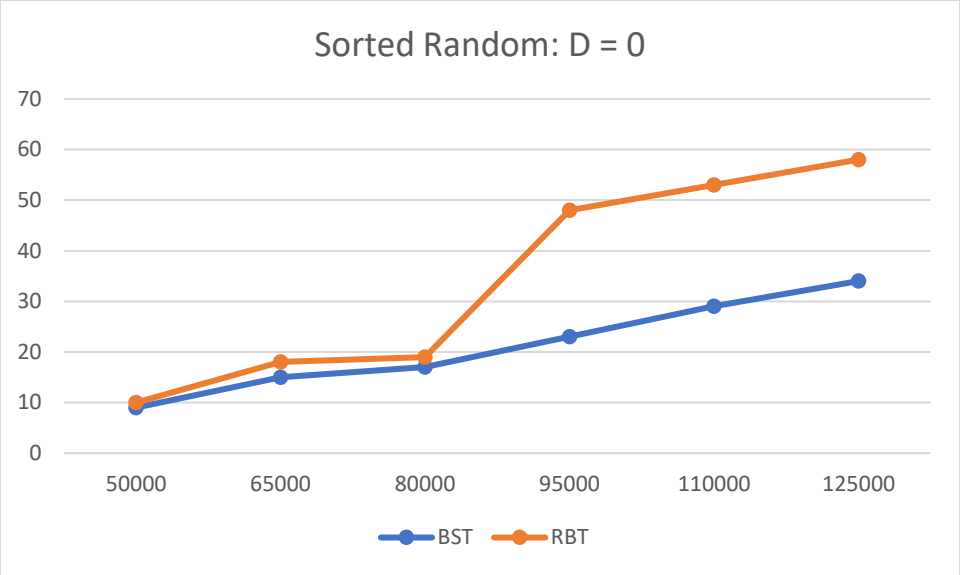
1. **Binary Search Tree and Red-Black Tree:**
    - The runtime in ms for Binary Search Tree and Red-Black Tree are as follows:

	Descending Order: D = -1		**Random Order D = 0**		**Ascending Order D = 1**	
Size (n)	**BST (ms)**	**RBT (ms)**	**BST (ms)**	**RBT (ms)**	**BST (ms)**	**RBT (ms)**
50000	3275	7	9	10	3268	7
65000	6084	10	15	18	5784	9
80000	9474	13	17	19	8478	12
95000	13233	26	23	48	12744	14
110000	17978	16	29	53	17419	18
125000	25177	20	34	58	22721	20

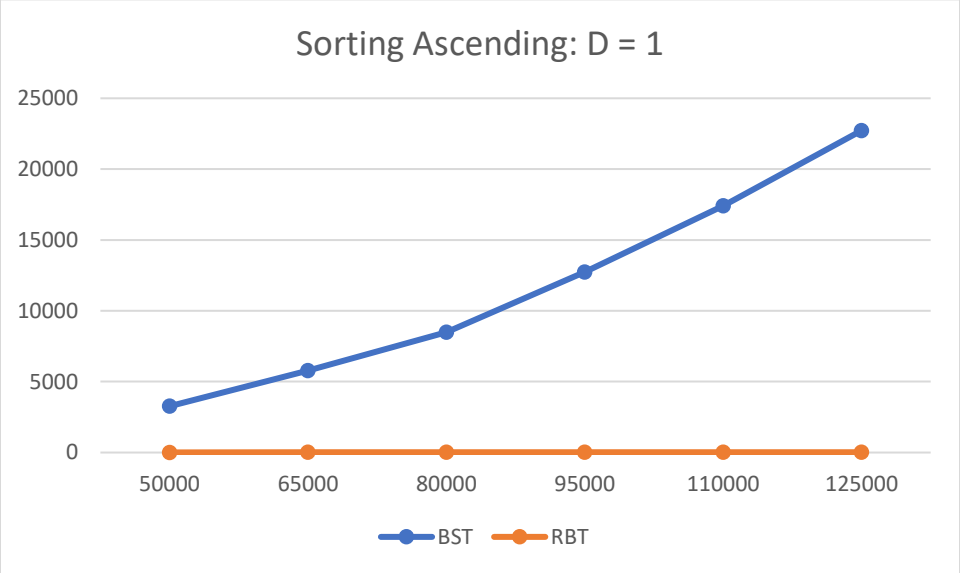
- The runtime taken by Binary Search Tree when the given array is in descending i.e., D = -1 is a lot greater as compared to time taken by Red-Black Tree.
- As the array size increase, the time taken by Binary Search Tree increases tremendously.
- For the same size of array, we can see Red-Black Tree takes very less time to sort and provide the expected result.
- We can say that Descending order is a worst-case scenario for Binary Search Tree, and it becomes very difficult to perform the Inorder Tree Walk.
- To Visit each node and return the value of each node takes a lot of time.
- Below is the comparison of the time taken by Binary Search Tree and Red-Black Tree when the array is reverse order i.e., D = -1.



- From the above figure, we can say that Red-Black Tree takes very less time even though the input size increases as compared to Binary Search Tree.
- The input size vs Runtime for the Random order is given below.



- From the above figure, we can see 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
- The input size vs Runtime for the Ascending order is given below.



- 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.

Average values of counters and duplicates:

- The below readings show 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.

	Direction D = -1						
Size (n)	**Counter duplicates BST**	**Counter duplicates RBT**	**Counter Case 1**	**Counter Case 2**	**Counter Case 3**	**Left Rotate**	**Right Rotate**
50000	0	0	49966	0	49971	0	49971
65000	0	0	64961	0	64971	0	64971
80000	2	0	79965	0	79970	0	79970
95000	2	2	94962	0	94970	0	94970
110000	0	3	109961	0	109969	0	109969
125000	4	0	124963	0	124969	0	124969

	Direction D = 0						
Size (n)	**Counter duplicates BST**	**Counter duplicates RBT**	**Counter Case 1**	**Counter Case 2**	**Counter Case 3**	**Left Rotate**	**Right Rotate**
50000	0	0	25657	9778	19315	14590	14503
65000	0	0	33325	12609	25260	18867	19002
80000	2	3	40957	15513	31069	23293	23289
95000	3	2	48854	18509	36952	27745	27687
110000	0	3	56621	21473	42834	32094	32213
125000	3	4	64111	24229	48400	36536	36513

- Similarly, when the array is sorted, there are 0 instances for case 2 and right rotation for Red-Black Tree.

	Direction D = 1						
Size (n)	**Counter duplicates BST**	**Counter duplicates RBT**	**Counter Case 1**	**Counter Case 2**	**Counter Case 3**	**Left Rotate**	**Right Rotate**
50000	0	0	49966	0	49971	49971	0
65000	0	0	64961	0	64971	64971	0
80000	0	0	79965	0	79970	79970	0
95000	0	0	94962	0	94970	94970	0
110000	0	0	109961	0	109969	109969	0
125000	0	0	124963	0	124969	124969	0

3. Discussion

Running time:

- The Binary Search Tree is a balanced search tree. Height of the binary search tree becomes log(n), so the time complexity of Binary Search Tree operations = O(logn).
- In each iteration or in each recursive call, the search gets reduced to half of the array.
- Time taken by Binary Search tree in case of Ascending order and Descending order is very high and can be considered as worst-case scenario.
- In case of Random order, Binary Search Tree is can be considered efficient as it takes less amount of time.
- The Red-Black tree has an average time complexity of O(1) and in the worst-case, the complexity becomes O(logn).
- Red-Black trees make less structural changes to balance themselves which makes them faster and more efficient in case of insert and delete.
- Red-Black tree is also called as self-balancing binary search tree. The time taken by Red-Black tree is very less as compared to the binary search tree and is quite efficient.

Limitations:

- Binary Search tree follows the recursive approach and because of that, it requires more stack space.
- Binary Search tree algorithm is also considered as error prone and difficult.
- Caching is very poor in Binary Search tree as it supports the random access of data.
- Red-Black tree is very complicated to execute.

Improvements:

- Binary Search tree implementation can be improved for the ascending order data. We can convert the Binary Search Tree into a height balanced binary tree or self-balancing binary tree. This will improve the traversal time on the new Binary search tree.
- Caching the nodes can also improve the running time of the binary search tree.

4. Conclusion:

From the results which are present above, we can say that Binary Search Tree is a little efficient as compared to Red-Black trees when it comes to Random order case. However, when it comes to ascending order and descending order cases, time taken by binary search tree is very high as compared to Red-black trees and it can be considered as a worst-case scenario. As the input size rises, the sorting time also increases rapidly in case of Binary Search Tree. Also, we can say that the chances of having duplicate values is more in case of random array as compared to ascending or descending order.