Stevens Institute of Technology

Department of Computer Science

CS590: ALGORITHMS

Prof. Iraklis Tsekourakis

Homework Assignment 3

Submitted by: **Malav Shastri**

CWID**: 10456244**

Introduction:

The Given problems are about binary search tree and Red Black Trees. In question 1 we are supposed to create a BST according to the implementation of the given Red Black Tree apart from the function deletion. We are also supposed to handle duplicate values while insertion in both the trees. The second question is all about storing back the sorted values from BST and RBT back to an array and that array is then passed to check_sorted() function which checks whether the array is sorted or not. Basically for this question we are taking the benefit of the in order tree walk. In order tree walk in both the BST and RBT ensures the sorted sequence of input elements. And by making a simple change in that, i.e. rather than printing the values simply store it in array. We are ensuring the sorted array. We are also supposed to get the number of elements copied in the array or let's say the size of the new array.

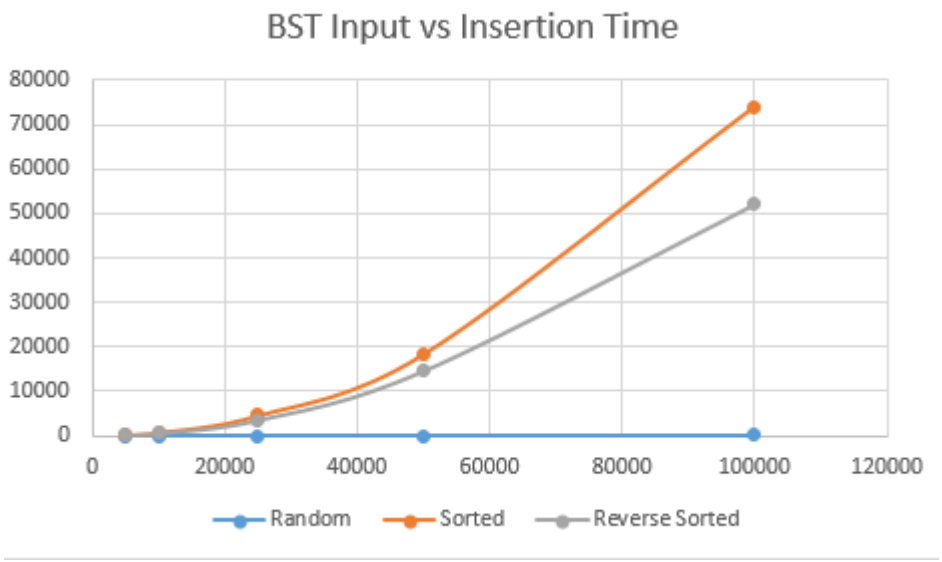
Third question is about different counters we know that RBT for the sake of self-balancing does different operations there are different cases for insertion process which again calls different operations like left rotate and right rotate. 3rd question is all about storing and printing that values 3 for insertion cases happens in insertion fixup and 2 for left rotate and right rotate. Apart from that we also need to print the counter for duplicate values. Question 4 is about the property no. 5 of RBT. Which is from any given node to all the available paths towards leaf node number of black nodes are same in all of those. For implementing this we are using recursion here. Just like concept of tree traversing us first traverse through the left most node of the tree we check whether its color is black if it is then we increase the black left height. The same height is return to the recursive call which traverses for the right sub tree.

The fundamental purpose of this problem is about how self-balancing trees and normal binary search trees behaves compared to each other. While RBT always maintains a height of O(log n) for all the cases. It does not matter whether the array is sorted, random or reverse sorted because it is the self-balancing tree. On the other hand BST is having a height of O(log n) for random input arrays but for sorted and reverse sorted height is going to increase in order of O(n). Below is the various analysis of BST and RBT for different input size and directions.

Table 1. Binary Search Tree Insertion Performance for various input sizes. With duplicate Counter values

Binary Search Tree						
N	**d=0** (Random)	**Duplicate Counter**	**d=1**(Sorted)	**Duplicate Counter**	**d= -1**(Reverse Sorted)	**Duplicate Counter**
5000	2	0	180	0	140	0
10000	5	0	750	1	580	0
25000	13	0	4600	1	3542	0
50000	30	1	18400	1	14640	0
100000	65	3	74000	1	52144	0
250000	215	15	>5 mins	>5 mins	>5 mins	>5 mins
500000	540	50	>5 mins	>5 mins	>5 mins	>5 mins
1000000	1280	230	>5 mins	>5 mins	>5 mins	>5 mins
2500000	3800	1468	>5 mins	>5 mins	>5 mins	>5 mins
5000000	9460	5772	>5 mins	>5 mins	>5 mins	>5 mins

Figure 1. Binary Search Tree Input vs Insertion Time graph for all 3 directions

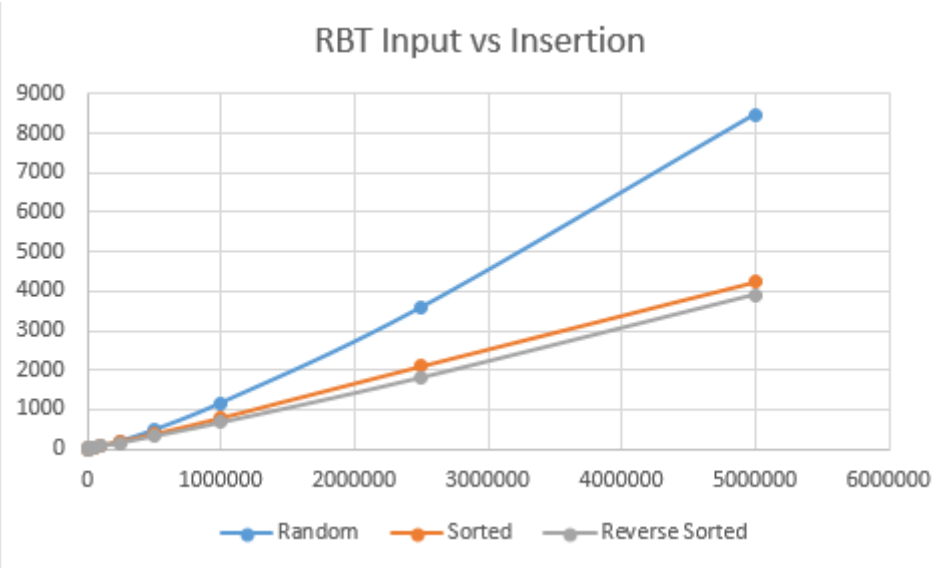


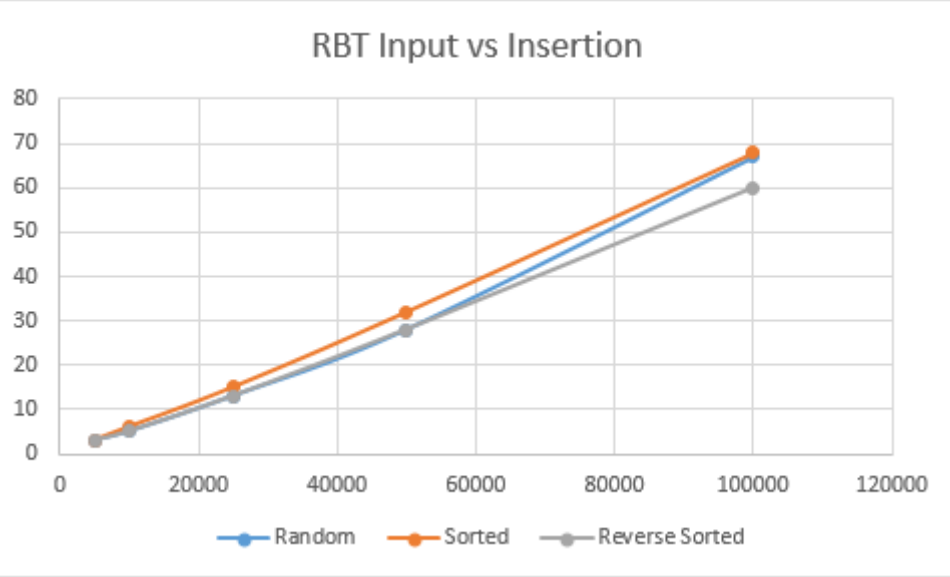
- Here in the graph you can see that as for random inputs the size of binary tree is O(log n) the insertion time is significantly linear compared to other two cases where height reaches to O(n).
- Also duplicate values more likely to occurred in random case. Duplicate values also changes insertion time a bit as you can see for reverse sorted and sorted case

Table 2. Red Black Tree Insertion Performance for various input sizes. With duplicate Counter values

Red Black Tree						
N	**d=0** (Random)	**Duplicate Counter**	**d=1**(Sorted)	**Duplicate Counter**	**d= -1**(Reverse Sorted)	**Duplicate Counter**
5000	3	0	3	0	3	0
10000	5	0	6	0	5	0
25000	13	0	15	0	13	0
50000	28	0	32	0	28	0
100000	67	4	68	0	60	0
250000	196	13	184	0	153	0
500000	480	52	369	0	318	0
1000000	1160	242	777	0	665	0
2500000	3580	1390	2088	0	1800	0
5000000	8470	5800	4228	0	3900	0

Figure 2. Binary Search Tree Input vs Insertion Time graph for all 3 directions





- Here for Red Black Tree I have taken two graphs just to show that insertion running time for all 3 are same but here as we are dealing with duplicate values which are occurring significantly more for Random case after input size passes 250000 the line for random case in graph 1 seems to increase faster than other two. In reality it's something like graph 2 where I have only considered the smaller input values for which number of duplicates are somewhat similar. The reason behind this is because as Red Black tree is balanced it has a height of O(log n) for all the cases.

Table 3. Red Black Tree different counter values for Random inputs.

Red Black Tree (Random - d=0)					
N	**Case 1**	**Case 2**	**Case 3**	**Left Rotate**	**Right Rotate**
5000	2539	991	1965	1466	1490
10000	5120	1956	3860	2909	2907
25000	12847	4946	9691	7276	7361
50000	25657	9754	19432	14559	14627
100000	51190	19353	38789	28923	29219
250000	128257	48765	97181	73078	72868
500000	256675	96987	193730	145384	145333
1000000	513161	194021	388250	291343	290928
2500000	1282749	484849	970949	727740	728058
5000000	2564272	969199	1940911	1455409	1454701

Table 4. Red Black Tree different counter values for Sorted inputs.

Red Black Tree (Sorted - d=1)					
N	**Case 1**	**Case 2**	**Case 3**	**Left Rotate**	**Right Rotate**
5000	4973	0	4978	4978	0
10000	9971	0	9976	9976	0
25000	24968	0	24973	24973	0
50000	49966	0	49971	49971	0
100000	99964	0	99969	99969	0
250000	249961	0	249967	249967	0
500000	499959	0	499965	499965	0
1000000	999957	0	999963	999963	0
2500000	2499952	0	2499960	2499960	0
5000000	4999950	0	4999958	4999958	0

Table 5. Red Black Tree different counter values for Reverse Sorted inputs.

Red Black Tree (Sorted - d=-1)					
N	**Case 1**	**Case 2**	**Case 3**	**Left Rotate**	**Right Rotate**
5000	4973	0	4978	0	4978
10000	9971	0	9976	0	9976
25000	24968	0	24973	0	24973
50000	49966	0	49971	0	49971
100000	99964	0	99969	0	99969
250000	249961	0	249967	0	249967
500000	499959	0	499965	0	499965
1000000	999957	0	999963	0	999963
2500000	2499952	0	2499960	0	2499960
5000000	4999950	0	4999958	0	4999958

Conclusion:

- From the results we can see that for random input arrays. Binary Search Tree and Red Black Tree takes almost the same time despite of one being normal binary tree and other one a self-balancing tree. The reason

behind this is for binary search tree random input sequence is like a best case for which it takes O(log n) time to insert all the values. Red Black tree being a balanced tree takes the same O(log n) time for each case including random case.

- The insertion time for sorted and reverse sorted should be more or let's say should not be considered it as its best case and results for Binary Search Tree actually shows that. Red Black tree is also a binary tree and same should happen for that one also. But here again the property of self-balancing plays it's role and insertion time is same as for the best case of any normal binary tree. The complete comparison graph of all the cases for both the trees have been given below.
- For Counter values as we can see from above 3 tables. Everything is normal for sorted arrays. But for reverse sorted input case 2 and left rotate is not happening at all. The reason behind that is simple we know that case 1 and case 3 involves operations relates/for left sub tree and here our input is reverse sorted so that justifies the no values for case 2 and left rotate.
- For Sorted sequences case 2 and right rotate does not occurs. As the array is already sorted case 1 and 3 occurs much more along with right rotate.
- For reverse sorted number of Case 3 and right rotate are same as it should be.
- For sorted number of Case 3 and left rotate are same.
- For reverse sorted and sorted inputs number of all corresponding cases are similar.