Homework Assignment 3

Abstract

The time taken for basic operations on a binary search tree is proportional to the height of the tree. Such operations run in Θ(lg n) worst case time for a complete binary tree with n nodes. However, if a tree is a linear chain of n nodes, the same operation takes Θ(n) worst case time. We can't always guarantee that binary search trees are built randomly but we can design variations of binary search tree with good guaranteed worst-case performance on basic operations. One such variation is the red-black trees. The main goal of the report is to analyze the running time of binary search tree and red-black tree. We'll be using different input sizes ranging from 50000 to 125000 with differently ordered inputs to check the performance of the operations. We'll be running a test at least 7 times to compute the average running time of an operation and the average counter values for duplicates and different insertion cases of red-black trees for each combination of input and size n.

Introduction:

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties:

- The value of the key to the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key to the right sub-tree is greater than or equal to the value of its parent (root) node's key.

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other path, so that the tree is approximately balanced. The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red- black tree with n keys, take O (lg n) time

We have used different input sizes n = 50000, 65000, 80000, 95000, 11000, 125000 and random, sorted and reverse sorted inputs to check the performance of these trees for different combinations.

We'll present and analyze the results of the two trees in the next section and check how well they perform. We'll also compare the running time of the trees with the best case and worst-case time complexities for them and see how much the results deviate from the standard time complexities.

Results and Evaluations:

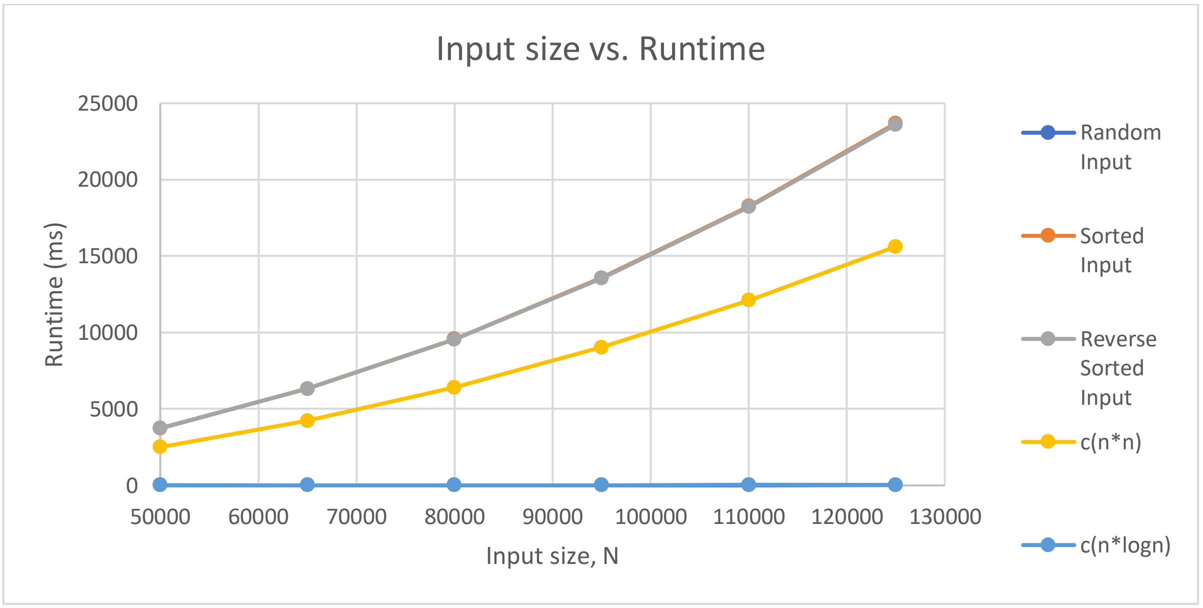
The average runtime of binary search tree for different parameters is described in the table below.

	Random Input		Sorted Input		Reverse Sorted Input	
N	**Runtime**	**Duplicates**	**Runtime**	**Duplicates**	**Runtime**	**Duplicates**
50000	9	1	3737.5	0	3729	0
65000	12	1	6315	0	6319	0
80000	15.5	2	9576.5	0	9555	0
95000	18	3	13560.5	0	13548.5	0
110000	22	3	18249	0	18218	0
125000	25	5	23687.5	0	23609	0

Table 1: Binary Search Tree Runtime in ms (Average Summary)

The running time of binary search tree for insertion and traversal for random input has a time complexity of O (n*lg n). The running time for sorted and reverse sorted input, however, has a time complexity of O (n*n). This is because the tree becomes a linear chain of nodes for sorted and reverse sorted inputs and hence takes a longer time for tree traversal.

Below is the Input Size vs. Runtime comparison for different sorted inputs along with the worst-case time complexity of binary search tree.



Graph 1: Input size vs Runtime for Binary Search Tree

From the above results, the binary search tree performs well for random input for different input sizes. However, the running time for the tree increases exceptionally for sorted and reverse sorted array because of skewed tree formation during insertion and hence the running time increases greatly. The running time for these trees also increase with the increase in the size of the input as the time complexity is directly proportional to the input size n. As we can see, the running time growth for BST agrees to the asymptotic notation for worst case time complexities.

The duplicates are present only for random sorted input and not for sorted and reverse sorted input since the element exists only once for the sorted inputs.

The average runtime of red-black trees for different sorted inputs is described in the tables below.

Random Input							
N	**Runtime**	**Case 1**	**Case 2**	**Case 3**	**Left Rotate**	**Right Rotate**	**Duplicates**
50000	10	25597	9636	19290	14499	14427	1
65000	13	33505	12688	25294	18941	19041	1
80000	16	41005	15637	31096	23342	23391	1
95000	20	48839	18638	37062	27861	27839	2
110000	23	56420	21161	42732	31936	31957	5
125000	27	64101	24295	48622	36370	36547	3

Table 2: Red Black Tree Runtime for Random Input in ms (Average Summary)

Sorted Input							
N	**Runtime**	**Case 1**	**Case 2**	**Case 3**	**Left Rotate**	**Right Rotate**	**Duplicates**
50000	9	49966	0	49971	49971	0	0
65000	12	64961	0	64971	64971	0	0
80000	13	79965	0	79970	79970	0	0
95000	16	94962	0	94970	94970	0	0
110000	20	109961	0	109969	109969	0	0
125000	21	124963	0	124969	124969	0	0

Table 3: Red Black Tree Runtime for Sorted Input in ms (Average Summary)

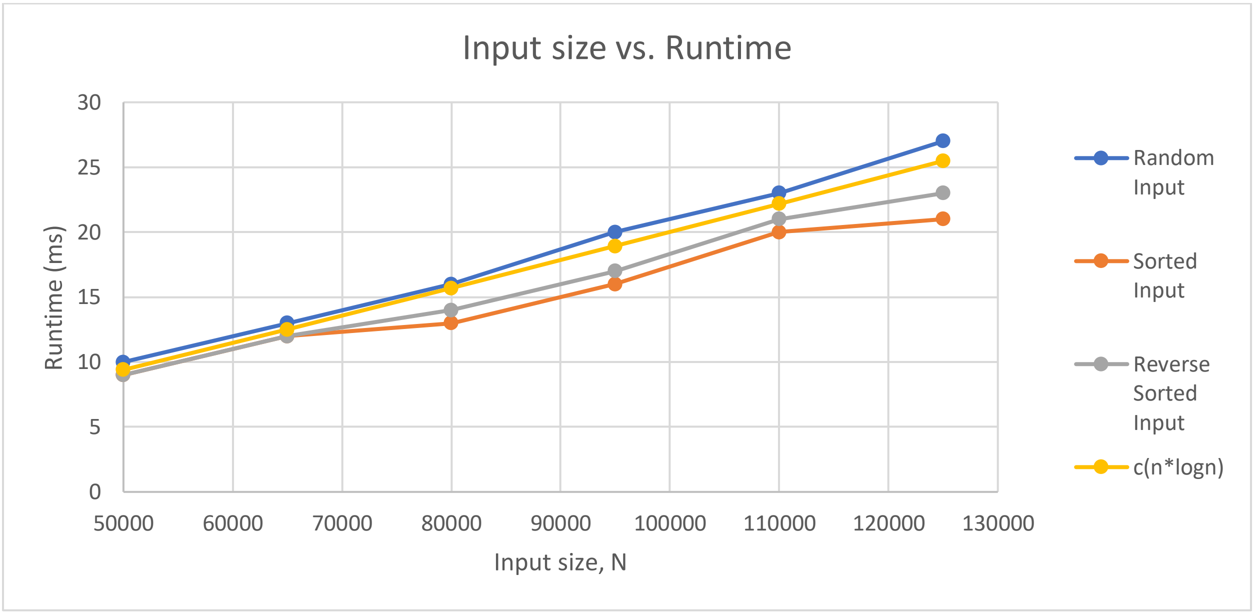
The running time of red-black tree for random input is almost equal to the running time for sorted and reverse sorted inputs since the red-black tree is balanced. Here the duplicates are also present only for random sorted input and not for sorted and reverse sorted input. The increase in counter numbers just slightly increases the running time of the operation since both LEFT-ROTATE AND RIGHT-ROTATE run in O(1) time.

Reverse Sorted Input							
N	**Runtime**	**Case 1**	**Case 2**	**Case 3**	**Left Rotate**	**Right Rotate**	**Duplicates**
50000	9	49966	0	49971	0	49971	0
65000	12	64961	0	64971	0	64971	0
80000	14	79965	0	79970	0	79970	0
95000	17	94962	0	94970	0	94970	0
110000	21	109961	0	109969	0	109969	0
125000	23	124963	0	124969	0	124969	0

Table 4: Red Black Tree Runtime for Reverse Sorted Input in ms (Average Summary)

The cases and rotation counters differ for the different sorted inputs as different actions are being performed to balance the tree in all the three cases. For sorted input, the binary tree becomes right skewed on insertion and hence we need to perform the left rotate operations to balance out the tree. Similarly, for reverse sorted order, the tree upon insertion becomes left skewed and hence we perform the right rotate operations to balance the tree. For random input, we need to perform both the right rotate and left rotate operation to balance the tree.

Below is the Input Size vs. Runtime comparison for different sorted inputs along with the worst-case time complexity for red-black tree.



Graph 2: Input size vs Runtime for Red-Black Tree

As we can see, the running time for random, sorted and reverse sorted input is comparable since the tree balancing operations are performed at each insertion. The worst-case running time for insertion and traversal of red-black trees should be comparable to O (n*lg n). Here, the running time growth for RBT agrees to the asymptotic notation for worst case time complexities.

Conclusion:

We're successfully able to test the running time for binary search tree and red-black tree for different input sizes and input order. From the above results, we can conclude that red-black tree is the more efficient tree when the input is in a sorted order. For random input, the red-black tree performs somewhat like the binary search tree.

The above test results agree with the time complexity for the binary search tree and red-black tree and the behavior is as expected. With small values of N, N and log(N) are comparable, but as the data grows large, logarithmic time quickly takes the lead. This shows how a Red-Black tree can make a significant improvement in searching, inserting, and deleting operations.