Sparsh Oza

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

1. Abstract:

A new Binary Search Tree must be implemented using the Red-Black Tree implementation that we already have. We need to handle the duplicate values while also updating the insertion process. In the event that a duplicate value is received, we must not insert it. For the Binary Search Tree and Red-Black Tree, we also need to change the Inorder Tree Walk algorithm so that it will traverse the tree and copy its elements back to the array in sorted order. It is not advisable to copy duplicates since they have already been discarded. Finally, we must return the total number of elements transferred 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.

Last but not least, we must tally and report the total number of Black nodes that are reachable along the path. For every scenario, we must evaluate the runtime behavior for various input sizes and present a report and analysis of the results.

2. Result:

Testing sorting algorithms

To complete the given experiment, two different types of sorting algorithms were used, and the time complexity of each one was assessed. These are the two algorithms:

- Binary Search Tree
- Red-Black Tree

Binary Search Tree:

The node-based binary tree data structure known as the "Binary Search Tree" has the following characteristics:

- 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 a value that go with it. The required key is compared to the keys in BST when searching, and if a match is made, the related value is returned. Always start your search at the root node. Search for the element in the left subtree if the data is smaller than the key value. If not, look for the element in the appropriate 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:

Inorder traversal is one approach to move through the Binary Search Tree. The left subtree is examined first during the in-order traversal process, then the root, and finally nodes on the right subtree. Beginning at the root, traversal proceeds to the left node, the left node once more, and so on until it reaches a leaf node. When all of the nodes have been visited, we copy that value into the array and proceed as before.

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:

Self-balancing Binary Search Tree is another name for the Red-Black Tree. Along with the key and value, each node also holds a third field called color. In this tree, Red and Black are the only colors that are stored. The balance of the tree is maintained during insertion and deletion operations thanks to the employment of these hues. These characteristics make it faster and less time-consuming to search than the Binary Search Tree. The RedBlack Tree has the following characteristics:

- 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 RedBlack 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:

Inorder traversal is a way to move through the Red-Black Tree. The left subtree is examined first during the in-order traversal process, then the root, and finally nodes on the right subtree. Beginning at the root, traversal proceeds to the left node, the left node once more, and so on until it reaches a leaf node. When all of the nodes have been visited, we copy that value into the array and proceed as before.

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:

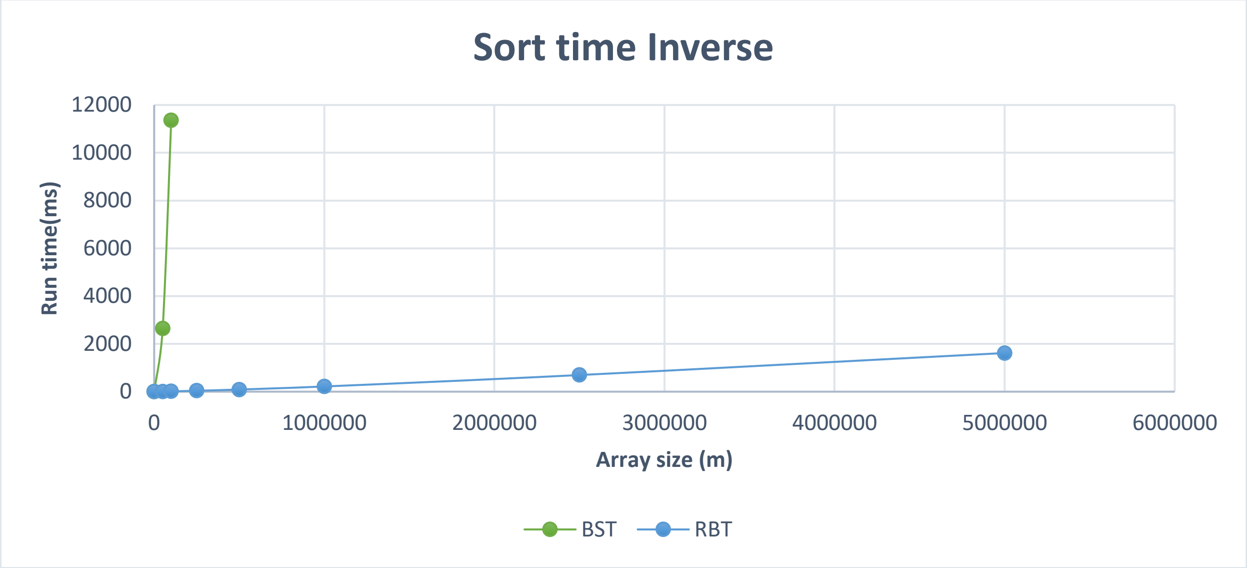
- The runtime in ms for Binary Search Tree and Red-Black Tree are as follows:

	Inverse		Random		Sorted	
n	BST	RBT	BST	RBT	BST	RBT
0	0	0	0	0	0	0
50000	2643.9	5.5	9	9.1	2688.2	6.1
100000	11358.4	13.3	19.5	21.5	10022.7	14.1
250000		38.6	66.5	68.7		41.3
500000		89	192.1	225.3		89.9
1000000		217.5	547.7	536.6		222.4
2500000		696	1893.9	2147.1		699.4
5000000		1617.2	5192.9	5236		1677.4

The runtime taken by Binary Search Tree when the given array is in descending is a lot greater as compared to time taken by Red-Black Tree.

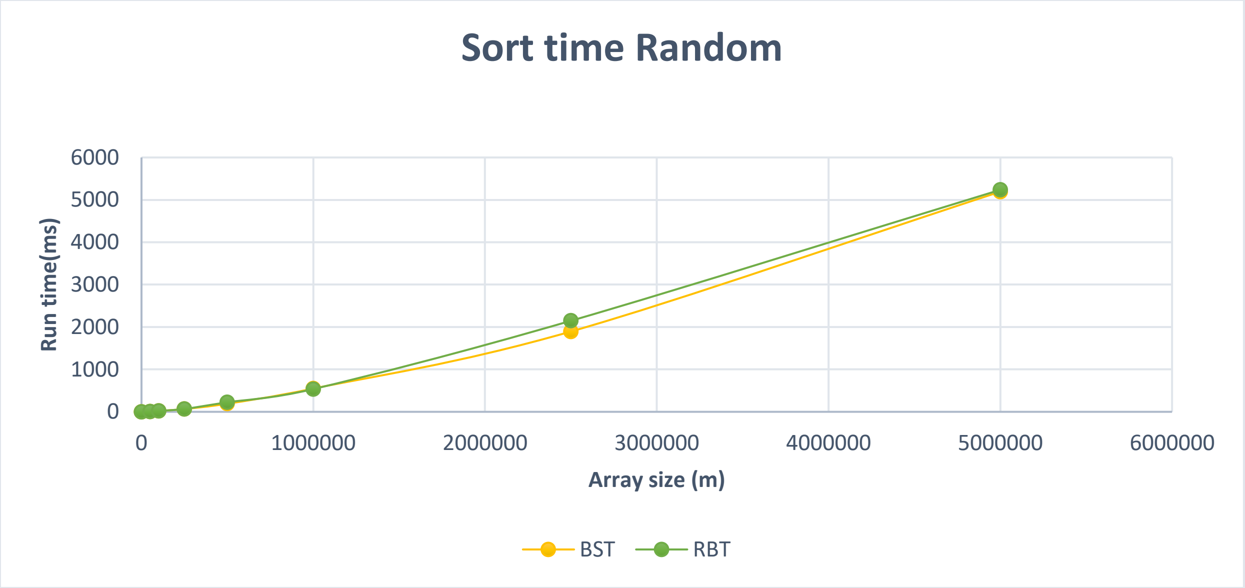
- The time required by Binary Search Tree significantly increases as the array size increases.
- Red-Black Tree sorts an array of the same size much more quickly and produces the desired results.
- Descending order is the worst-case scenario for a Binary Search Tree, and the Inorder Tree Walk becomes quite challenging.
- It takes a long time to visit each node and return its value.
- The time taken by Red-Black Tree and Binary Search Tree when the array is in reverse order

Sparsh Oza


Sort time Inverse
12000
10000
8000
6000
4000
2000
0
Run time(ms)
0
1000000
2000000
3000000
4000000
5000000
6000000
Array size (m)
BST
RBT

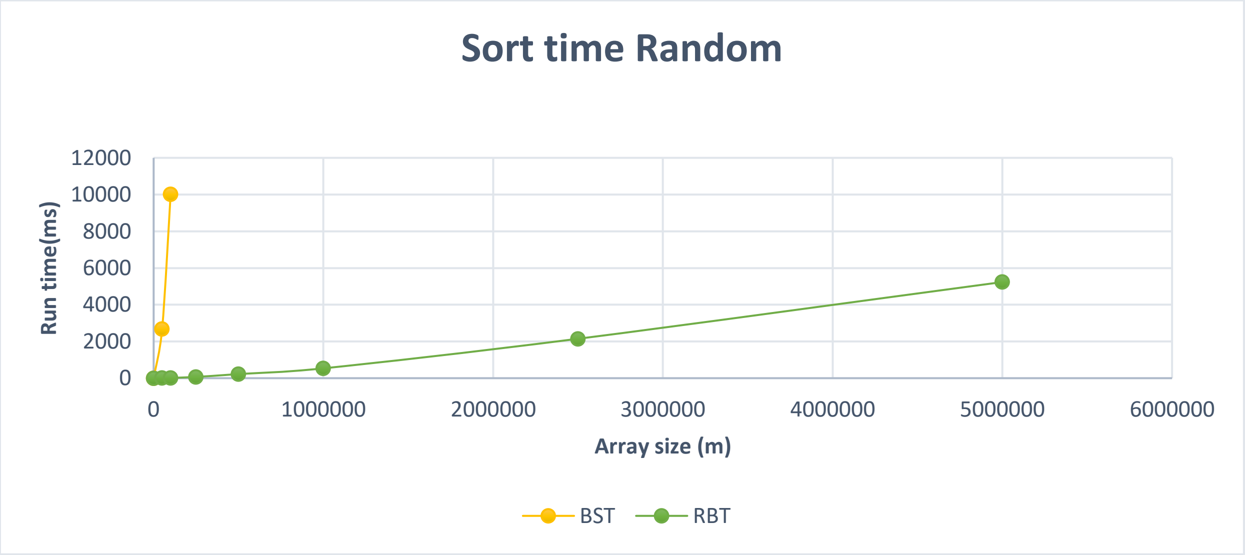
From the above figure, we can say that Red-Black Tree takes very less time even though the input size increases as compared to Binary Search Tree.

The input size vs Runtime for the Random order is given below.



- As shown in the above figure, the Red-Black Tree roughly takes the same amount of time when dealing with random sorted arrays, but as the input size increases, we can observe a jump in the Red-Black Tree's processing time.
- 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



- When dealing with Sorted order arrays, a Binary Search Tree's traversal of the data takes a long time. Similar to the descending order case, it can also be seen as the worst-case scenario.
- The Red-Black Tree is far more efficient than the Binary Search Tree and requires much less time even as the size of the input rises.
- For Binary Search Tree, as the input size increases, the time taken by the algorithm also increases.

Average values of counters and duplicates:

- The average values of the counters for all possible combinations of input size and orientation are displayed in the readings below.
- There are 0 instances of case 2 and left rotation for Red-Black Tree since the array is reverse-sorted or when it is sorted.

	Inverse						
n	Counter Duplicates BST	Counter Duplicate RBT	Case 1 Counter	Case 2 Counter	Case 3 Counter	Left Rotate	Right Rotate
0	0	0	0	0	0	0	0
50000	0	0	49966	0	49971	0	49971
100000	0	0	99964	0	99969	0	99969
250000		0	249961	0	249967	0	249967
500000		0	499959	0	499965	0	499965
1000000		0	999957	0	999963	0	999963
2500000		0	2499952	0	2499960	0	2499960
5000000		0	4999950	0	4999958	0	4999958

	Random						
n	Counter Duplicates BST	Counter Duplicate RBT	Case 1 Counter	Case 2 Counter	Case 3 Counter	Left Rotate	Right Rotate
0	0	0	0	0	0	0	0
50000	0.6	0.9	25670.6	9693.4	19418.9	14548	14564.3
100000	4.2	2.3	51349.8	19409.3	38832.3	29116.7	29124.9
250000	16.2	16.4	128384.7	48460.4	97020.1	72712.8	72767.7
500000	59.3	65	256667	97088.4	194182.7	145559	145712.1
1000000	233.7	238.2	513267.4	194036.1	388275.1	291189.2	291122
2500000	1459.5	1463.9	1282898.7	484826.3	970133	727406.4	727552.9
5000000	5847.3	5840.7	2564236	969799.1	1939528.1	1454801.7	1454525.5

	Sorted						
n	Counter Duplicates BST	Counter Duplicate RBT	Case 1 Counter	Case 2 Counter	Case 3 Counter	Left Rotate	Right Rotate
0	0	0	0	0	0	0	0
50000	0	0	49966	0	49971	49971	0
100000	0	0	99964	0	99969	99969	0
250000		0	249961	0	249967	249967	0
500000		0	499959	0	499965	499965	0
1000000		0	999957	0	999963	999963	0
2500000		0	2499952	0	2499960	2499960	0
5000000		0	4999950	0	4999958	4999958	0

3. Discussion

Running time:

- A balanced search tree is the binary search tree. The temporal complexity of binary search tree operations equals O because the height of the binary search tree becomes log(n) (logn).
- The search is scaled down to cover only half of the array with each iteration or recursive call.
- The time required by the Binary Search tree under both ascending and descending order conditions is extremely high and can be viewed as the worst-case situation.
- Binary Search Tree can be regarded as efficient in the situation of random order because it requires less time.
- The Red-Black tree's time complexity averages out to O(1), and in the worst case, it rises to O. (logn).
- Red-Black trees balance themselves with less structural changes, which makes them quicker and more effective for insert and delete operations
- Self-balancing binary search tree is another name for the Red-Black tree. When compared to a binary search tree, the RedBlack tree takes much less time and is extremely effective.

Limitations:

- Binary Search trees use a recursive approach, which means that more stack space is needed.
- The binary search tree approach is regarded as challenging and error-prone.
- Because Binary Search Tree provides random access to data, caching performance is quite bad hence causing segmentation errors for over 250,000.
- The Red-Black tree is extremely difficult to implement.

Improvements:

For the data in ascending order, the implementation of the binary search tree can be improved. The Binary Search Tree can be transformed into a height-balanced or self-balancing binary tree. The new Binary search tree's traversal time will be improved as a result. Another way to decrease the binary search tree's processing time is to cache the nodes.

4. Conclusion:

According to the aforementioned data, Binary Search Trees are marginally more effective than Red-Black trees in the case of Random order. However, binary search trees take far longer than Red-black trees to process situations involving ascending and descending order, so this can be viewed as the worst-case scenario. In the case of Binary Search Tree, the sorting time rapidly increases as the input size increases. Additionally, we can state that compared to ascending or descending order, a random array has a higher likelihood of containing duplicate items.