CS 590 – B HW2 Analysis of Binary Search Trees and Red Black Trees

Report by: Sanjeet Vinod Jain

# Testing Hardware used:

CPU: Ryzen 5600 6 core 12 thread processor at 4.0Ghz

RAM: 16gb 3200mhz

Storage: NVME SSD 2000MB/s read-write speed

Operating System: Windows 11 with Virtualized Linux Bash

# Testing Parameters:

Each algorithm was run with varying inputs that includes random, ascending sorted, and descending sorted array inputs of size m = 50000 100000 250000 500000 1000000 2500000 5000000

Each combination of m was allowed to run for a maximum of 6 mins after which it was automatically timed out.

**All scripts to run the code were ran parallel to finish the project on time**

# Introduction:

I have implement a new Binary Search Tree based on the given implementation of Red Black Trees. the insertion routine has been updated to handle the duplicate values. If the duplicate value arrives, then it will not insert that duplicate value. The Inorder Tree Walk algorithm for Binary Search Tree and Red-Black Tree has been modified such that it will traverse the tree and copy its elements back to the resulting array in a sorted order. Duplicates are not be copied as they are eliminated while insertion. Furthermore, there are counters implemented for the following occurrences over the sequences of insertions.

the number of duplicates.

The number insertion done for specific cases for Red-Black Tree only.

The number of left rotate and right rotations done for Red-Black Tree only.

the number of Black nodes which are accessible on the path

Lastly, the runtime performance for different input sizes is recorded

# Failures and Limits Applied:

Binary Search tree failed at n greater than 250000 i.e ( 500000 1000000 2500000 5000000 ) due to limitations of c++ and memory availability for larger stack sizes

# Results:

## 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:

*Algorithm (TREE-INSERT(T,x))*

*y = NIL, x = T.root*

*while (x ≠ NIL) do*

*y = x*

*if (z.key < x.key) then*

*x = x.left*

*else x = x.right*

*od*

*z.p = y*

*if (y = NIL) then*

*T.root = z*

*else if (z.key < y.key) then*

*y.left = z*

*else y.right = z*

## 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:

*Algorithm (INORDER-TREE-WALK(x))*

*if (𝑥≠𝑁𝐼𝐿)then*

*INORDER-TREE-WALK(x.left)*

*print x.key*

*INORDER-TREE-WALK(x.right)*

*fi*

# 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:

### 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:

## Binary Search Tree and Red-Black Tree:

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

Table

Description automatically generated

The runtime taken by Binary Search Tree when the given array is in descending is greater than time taken by Red-Black Tree.

As the array size increase, the time taken by Binary Search Tree increases.

For the same size of array, Red-Black Tree takes very less time to sort.

We can say that Descending order is a worst-case scenario for Binary Search Tree as to Visit each node and return the value of each node takes a lot of time. And in this case fails after an input size of 250000 due to stack size limitations of C++

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.

From the above figure, we can see that in case of random sorted arrays, Red-Black Tree roughly takes same amount of time as a Binary Search Tree but as the input size increases, the time taken by the Red-Black Tree is slightly less than that of Binary Search Tree.

Red-Black Tree is slightly more efficient when it comes to very large Random arrays

In case of Ascending 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 like the descending order case.

The Red-Black Tree takes very less time and is very efficient for all input sizes as compared to Binary Search Tree.

For Binary Search Tree, as the input size increases, the time taken by the algorithm also increases. And in this case fails after an input size of 250000 due to stack size limitations of C++

## 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.

**Table

Description automatically generated**

Since the array is reversed sorted/descending sorted, there are 0 instances for case 2 and left rotation for Red-Black Tree.

**Table

Description automatically generated**

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

**Table

Description automatically generated**

# 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 = average case of 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 the order of O(n) when the binary search tree is skewed.

In case of Random order, Binary Search Tree is can be considered somewhat efficient as it takes less amount of time with an average case of O(logn).

The Red-Black tree has an time complexity of O(logn) for any case.

Rebalancing operation has an average time complexity of O(1) and worst time complexity of 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.

## Conclusion:

From the results which are present above, we can say that Binary Search Tree is a somewhat as 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.