Nikunj Pradhan

Self-Balancing Binary Search Trees

Abstract:

A Binary Search Tree (which will be referred to in the paper as a BST) is a data structure used to organize and store data in a sorted and directed manner. Each node in a BST has only two sub nodes or children, a left node, and a right node. The node child contains a value less than the parent node and the right child contains a value greater than the parent node. This abstract structure allows for a quick and efficient method for node creation, deletion, and searching.

However, a normal BST is useful in the structure of the tree. The desired complexity of functions such as insertion, deletion, and searching should be O~log(n). When a binary search tree is unbalanced on its sides it becomes as efficient as a linear structure with a complexity of O~n. To prevent this from happening, a self-balancing BST must be implemented to keep the time complexity for operations to O~log(n). In this paper we will implement the Red-Black BST and compare its results with a normal BST in terms of efficiency and speed.

Requirements and Properties for Red-Black Trees:

In addition to the properties of a BST, the Red-Black Tree must follow these additional requirements:

1) Every node must either be red or black.
2) The root of the tree must be black.
3) Every leaf (Null) must be black.
4) There cannot be adjacent red nodes so every red node must have black children
5) All nodes to their descendent leaf nodes must have the same number of black nodes.

How is the Height of the Red-Black Tree Guaranteed O~log(n)?

Using property 5, the number of nodes from a root or sub-root to its most distant leaf cannot be more than twice the number of nodes to the closest descendant leaf from the root or sub-root. Thus, we can find the black height is if the height of the BST is D: black height >= D/2

For a standard BST, let X be the number of nodes of the tree and let p be the minimum number of nodes from the root to a leaf, so $X >= 2^p-1$ thus $p <= \log_2(X+1)$. Using property 5, a Red-Black Tree with X nodes, $\log_2(X+1)$ will be the most black nodes from the root to a leaf. Additionally, using property 3, being that all leaves must be black, at least $X/2$ of X nodes are black. Thus, in every Red-Black Tree with X nodes has height $<= 2\log_2(X+1)$ so a Red-Black BST guarantees O~log(n) efficiency.

Demonstration

Standard BST Deletion:

In a standard BST deletion function, there are three cases to consider:

1) Leaf Node Deletion: Directly remove the node.
2) Single Child Node Deletion: Remove the node and replace it with its child.
3) Two Children Node Deletion: Replace the node with its successor and then delete the node from its original position.

The general deletion algorithm:

Locate the Node: Move through the tree from the root to find the node to be deleted. This takes O(h) time, where h is the height of the tree.

Delete the Node:
Leaf Node: Simply remove the node from the tree.
Single Child: Replace the node with its child.
Two Children: Swap the node's value with its successor, then recursively delete the successor. Finding the successor requires finding the minimum node on the subtree on the right taking O(h) time.

Return the Tree: Recursively return the modified BST.
Red-Black BST Deletion:

In a Red-Black BST implementation it follows the same three cases as above however, rebalancing will also take place. After finding and deleting the node it will go through rebalancing and recoloring.

Rebalance the Tree:

Depending on the colors and positions of sibling nodes and other nodes in the tree, there are four cases for rebalancing the tree after deletion. After certain rotations and color adjustments, the BST will keep the O~log(n) complexity.

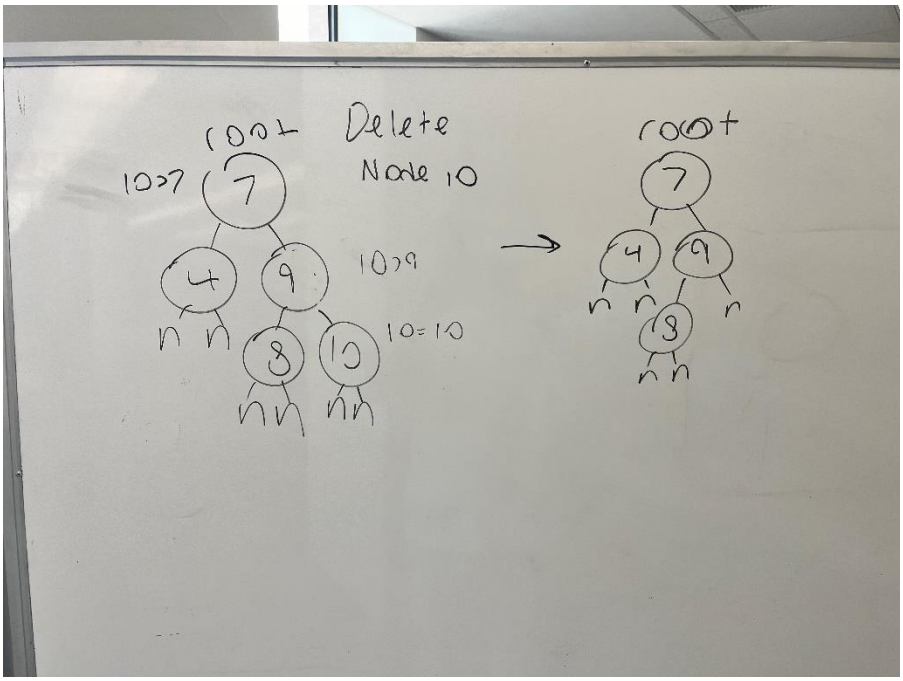
1. **Deleted Node is Red.** Perform normal Deletion method because a red node has no effect on the properties of Red-Black Tree.

2. **Sibling is Red.** Perform a rotation to switch the sibling and the parent's positions. Then recolor the sibling to black and the parent to red.

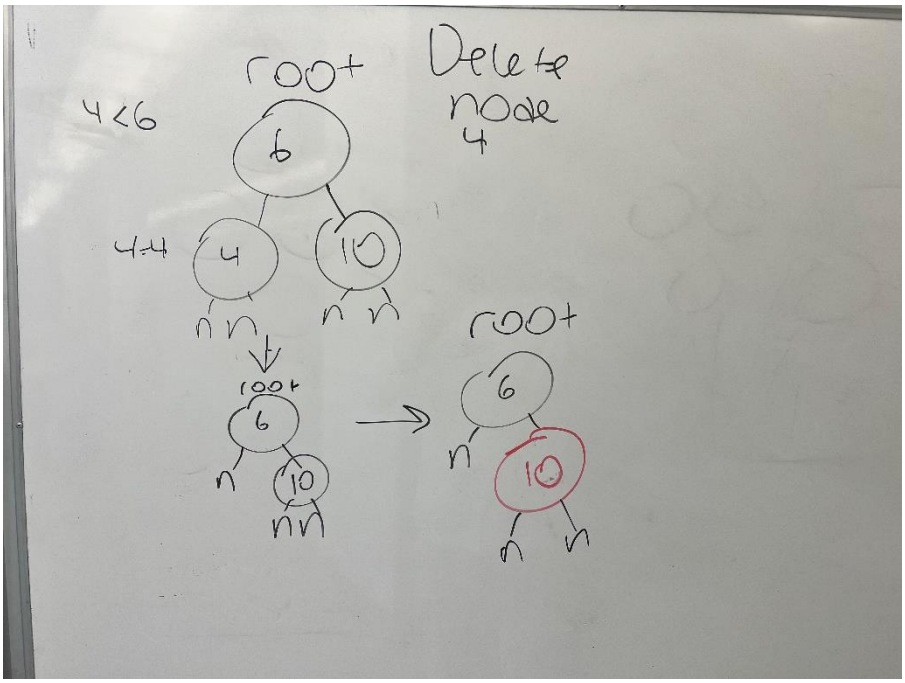
3. **Sibling and Sibling's Children are Black**. Recolor the sibling red. This shifts the issue of maintaining black height up the tree to a parent node. Where it will recursively fix the tree's black height as it moves to the root

4. **Sibling is Black with One Red Child**. Perform a rotation to move the red child that leads corrections that align the black height across the tree.

5. **Sibling is Black with Both Children Red or Distant Relative is Red.** Perform a rotation that moves the red child to the parent's position and change the color the new parent black.



In this example of an Unbalanced Tree. We want to delete node with the value 10, we compare it to the root and go right until we reach the node whose value is 10. Then we simply deallocate and delete the node. Node 9 now has a NULL in its right child.



In a Red Black Tree, we follow the same steps being that you find the node you want to delete. After deleting the node, it is replaced with a NULL on the left child of the root. However, since its sibling is black, and so are its children (NULLs are considered black) you can simply recolor the sibling as red to keep the properties of the Red Black Tree.

Performance Analysis

To test the efficiency of both an unbalanced BST and a Red-Black Tree, large benchmarks had to be set. To keep O~log(n) time complexity a Binary Search Tree remains useful, without rebalancing of the tree after insertions and deletions, it will slowly become a O~(n) complexity. A variety of input files were created that include both insertions and deletions, sorted input files, and reverse sorted files, along with a different number of operations performed.

Test Type	Red-Black BST (Height)	Unbalanced BST (Height)
Insert Only / 10,000 Operations / Unsorted	28	30
Insert Only / 8,000 Operations / Unsorted	25	30
Insert Only / 4,000 Operations / Unsorted	13	28
Insert Only / 10,000 Operations / Sorted	24	8642
Insert Only / 8,000 Operations / Sorted	22	5539
Insert Only / 4,000 Operations / Sorted	21	3274
Insert Only / 10,000 Operations / Reverse Sorted	24	8642
Insert Only / 8,000 Operations / Reverse Sorted	22	5539
Insert Only / 4,000 Operations / Reverse Sorted	21	3274
Insert and Remove / 5,000 Operations / Unsorted	13	29
Insert and Remove / 5,000 Operations / Sorted	19	1760
Insert and Remove / 5,000 Operations / Reverse Sorted	19	1749

Conclusion

In comparing Red-Black Trees to Unbalanced Binary Search Trees through these operational tests, the data clearly demonstrates the efficiency of Red-Black Trees in maintaining lower tree heights, especially in complex scenarios involving sorted and reverse-sorted data. What is particularly notable is the ability of a Red-Black Tree to handle extreme cases where unbalanced trees had significantly increased heights, such as in sorted and reverse-sorted operations. This capability demonstrates the advantage of Red-Black Trees, which perform essential rebalancing and recoloring to ensure an optimized performance through benchmarking.

This paper highlights the significant benefits of Red-Black Trees, especially for data structures that need high efficiency and stability. While unbalanced BSTs are simpler and work well for less complex tasks, Red-Black Trees offer a more reliable and scalable option, delivering consistent performance that maintains balance and ensures stability. Overall, this analysis clearly favors Red-Black Trees for environments where efficient tree operations are crucial for system performance.