Red-Black Trees

John Lindbergh

ABSTRACT

Red-black trees, invented by Rudolf Bayer in 1972, are a variety of self-balancing binary search trees. By maintaining "balance," a term that will be more concretely defined in section 3, the tree can guarantee a search time of O(log n). In this paper, this self balancing method will be outlined as well as its specific advantages over alternative self balancing tree algorithms. Furthermore, this paper aims to analyze the insertion efficiency, of both red black trees and AVL trees. To validate the conclusions drawn from the theoretical analysis, this paper finds an empirical analysis of insertion efficiency class of Red-Black Trees. To extend red-black trees to a real-world application, examples of their specific use cases will be explored.

1 INTRODUCTION

The purpose of Red-Black Tree, like any self-balancing tree, is to solve one of the weaknesses of a traditional binary search tree. When nodes are successively inserted into a BST rather than constructing the tree with a known median, the efficiency class of insertion and searching cannot be guaranteed. The following figure 1 illustrates this:

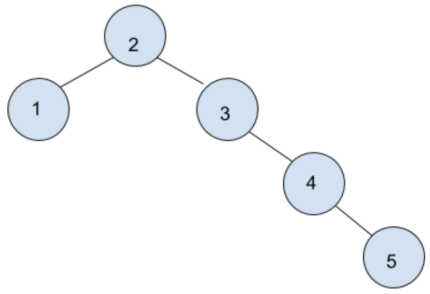

Figure 1: An unbalanced BST

Because the nodes were iteratively inserted, the algorithm naturally assumes the first node that was inserted (2) to be the root. As two is not the median value of the array, the number of nodes on its left and right are uneven. In this case there is one node on the left and three nodes on the right. This in turn causes the height of the tree to be higher than an otherwise balanced BST which should have a height of

$$\lceil log\ n \rceil .$$

For five nodes, a balanced BST should have a height of the ceiling of 2.32. Yet, the unbalanced tree has a height of four therefore this tree can no longer boast a log n efficiency class. Red-Black Trees are able to remedy this problem by ensuring approximate balance after every insertion. Each node is given an additional boolean attribute, "red," or "black," and the tree conforms to the following rules [1]:

1 Every undefined node that has a parent leaf node is colored black
2 If a node is red, than its children are black.
3 Every path in the tree from root to leaf should have the same number of black nodes.

2 INSERTION

On insertion into a non-empty tree, a node is colored red because otherwise the parity of a given path would be unchanged by a black node because the null leaf nodes are already colored black. It is inserted according to BST order which creates two initial cases: If the parent of the new node is black, the operation is complete because the tree is not in violation of Red-Black Tree order. If the parent of the new node is red, however, it has either unbalanced the tree, or the tree needs to be recolored, or both. To determine whether or not the tree is unbalanced, the sibling of the parent node of the new node otherwise known as its uncle is taken into account. Its coloring is indicative of the balance of the subtree because if it is colored black, and its sibling is red, the number of their descendants must be different. In the case that the new node's uncle is red, the tree must be recolored in the following manner [5]:

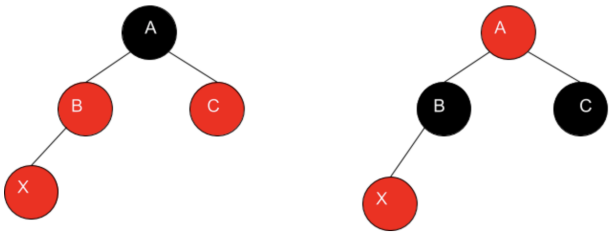

Figure 2: Recoloring Process

Rotation

In the case that the new node's uncle is black, a rotation must be performed. A right rotation is performed if the uncle is the right child of the grandparent, and a left rotation is performed if the uncle if the left child of the grandparent. Once the rotation has been performed, the nodes are recolored to conform to the rules of a Red Black Tree and the rebalancing

recurs from the root of the now balanced subtree to the root node of the tree [2]. The methods of rotation are as follows:

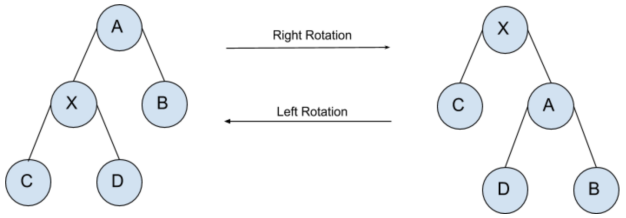

Figure 3: Rotation Process

Example Insertion

On the insertion of the 2-node as shown in figure 4, it is initially colored red. Because the tree is empty, the 2-node becomes the root node and is therefore recolored black. When the 1-node is inserted, it is colored red and inserted according to BST order. As the red child of a black node, it is not in violation of the red-black tree's properties and no further actions are required. The same process happens with the 4-node which does not commit any violations and prompts no further actions.

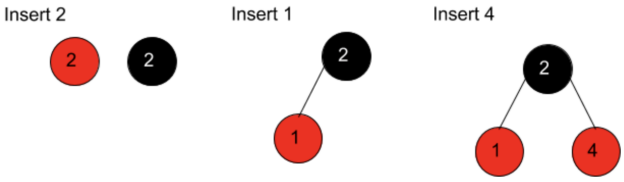

Figure 4: Insertion of 1, 2, and 4

When the 5-node is inserted in figure 5, however, the subsequent process becomes more complex. Insertion according to BST order places it as the right child of the 4-node. This violates the second property of red-black tree order because its parent node, the 4-node, is red, and the rule states that if a node is red, than its children are black. This violation causes the algorithm to check the color uncle of the inserted node. If it is colored black, the algorithm will not only recolor, it will also perform a rotation. Seeing as the uncle of the inserted node is red, only a recoloring of the parent nodes that recurs up the tree is necessary. Finally, the root node is returned to its original color as it is an exception to this rule.

The insertion of the 9-node in figure 6 is the first instance of the rotation process. It is initiated by a violation of the second property of red-black tree order as seen in the insertion of the 5-node. Because the color of the uncle node of the 9-node is null, it is inferred to be colored black in accordance

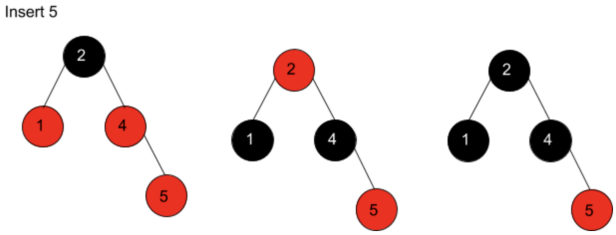

Figure 5: Insertion of 5

with the first property of red-black tree order. These two factors prompt not only a recursive recoloring of the tree, but also a rotation that recurs up the tree. In the final step of the insertion, we see a left rotation of the grandparent node of the 9-node. This rotation causes the 5-node to become the parent of the 4 and 9-nodes and take the place of the 4-node. This new placement is treated as another call to the insertion function that checks to see if any further actions are required.

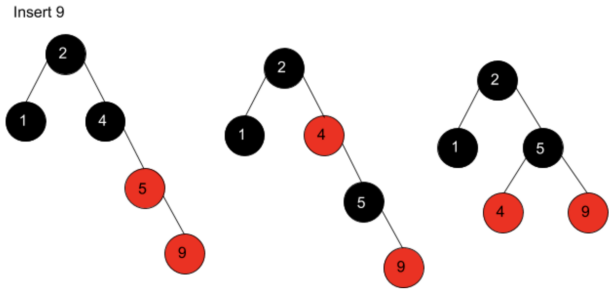

Figure 6: Insertion of 9

3 DEFINING BALANCE

RB trees are not as rigidly balanced as AVL trees, and while this may seem negligible, this can result in a height difference with an asymptotic relation

$$\frac{AVL}{RB} \simeq 0.72. \tag{1}$$

For this reason, red black trees are often called "approximately balanced" in textbooks. [3] Therefore, Red-Black trees do not perform well in instances in which lookup efficiency is a priority but insertion efficiency must be optimized.

4 EMPIRICAL ANALYSIS

The true strength of red-black trees comes from the efficiency of their insertion [3]. As a rather esoteric data structure, red-black trees have not been subjected to a mathematical analysis of their theoretical insertion efficiency like their more popular AVl counterparts. Moreover, some sources disregard

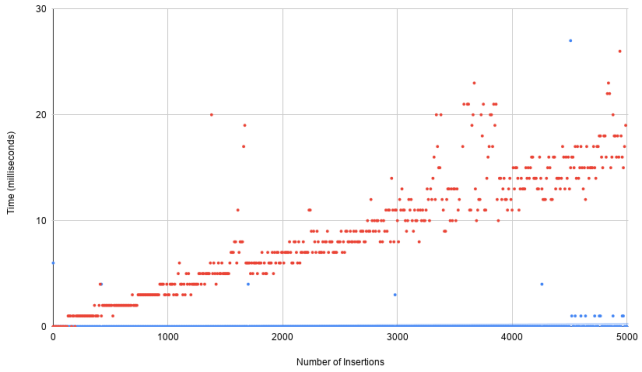

Figure 7: Empirical Analysis of Red-Black Trees in relation to AVL trees

their difference entirely. These reductionist analyses often recognize that red-black tree insertion is more efficient than AVL insertion but conclude with a worst case efficiency class of

$$\lceil log\ n \rceil .$$

This is erroneous because the consensus in studies of AVL trees is of the same oversimplified efficiency. One might encounter this false equivocation and determine that the stated disparity is negligible. Given the chaotic nature of both red-black and AVL tree insertion processes, an empirical analysis lends itself well to better understand their relative efficiency.

In this study, an AVL tree and red-black tree were created in C++ with identical node overhead. The insertion functions of each were analyzed by measuring the time elapsed from the point at which they were called to their completion. To get a accurate evaluation of their asymptotic efficiency, a set of 5000 nodes were inserted with uniform distribution at an interval of 10. This distribution was chosen because it more could more closely represent the distribution of realistic data and caused a more frequent unbalanced tree unlike a normal distribution in which the nodes were more frequently closer to the median of the set.

5 CONCLUSION

The result of the empirical analysis yielded a surprisingly vast disparity between red-black tree and AVL tree insertion efficiency. In figure 7, the data from the red-black tree and the AVL tree are shown in blue and red respectively. While the time data of the AVL tree closely follows a logarithmic curve, the red-black tree follows a near constant time curve. Further analysis shows that this curve is actually logarithmic, but with a fractional coefficient of approximately 1/30. This empirical analysis demonstrates that the asymptotic divergence from AVL insert efficiency is not negligible and further elucidates the realistic insertion efficiency of red-black trees.

REFERENCES

[1] John Morris, E.A., *Data Structures and Algorithms: 8.2 Red-Black Trees*, Retrieved October 21, 2019 from https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms: Red-Black Trees.*, Retrieved October 21, 2019 from http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap14.htm

[3] Ben Pfaff, *Performance Analysis of BSTs in System Software* Retrieved October 21, 2019 from https://benpfaff.org/papers/libavl.pdf

[4] Andrew W. Appel, *Efficient Verified Red-Black Trees* Retrieved October 21, 2019 from https://www.cs.princeton.edu/~appel/papers/redblack.pdf

[5] James Stewart, *CSC 378: Data Structures and Algorithm Analysis: Red-Black TREES* Retrieved October 21, 2019 from http://www.dgp.toronto.edu/~jstewart/378notes/16redBlack/