Group Project Report, “Red Black Tree”

Noah Mathew, Ethan Vowels, Ethan Pritchett, Nathan Garcia, and Rylan Kinzle

Data Structures

Professor Booth December 8, 2024

Introduction to Red Black Trees

To understand the logic behind a Red-Black Tree, one must first have a basic understanding of a Binary Search Tree. A Binary Search Tree (BST) is a typical binary tree with a couple of added rules. For every node, X, the values of the items in the left subtree are less than the item in X, and the values of the items in the right subtree are larger than the item in X [1]. A Red-Black Tree adds a couple more rules to the BST to have more efficient functions. Every node is either red or black, the root is always black, there may be no consecutive red nodes, and every path from a node to a null pointer contains

the same amount of black nodes. This allows for some special asymptotic performance. For the RBT, operations take O(log N) time in the worst case.

Design Decisions

The most crucial design decision we made for this project was to separate the Red-Black Tree (RBT) logic from the visualization of the tree. We developed a Red-Black Tree class that encapsulates the tree structure and its operations, offering a simple interface for interaction. This approach enabled our visualization team to focus on rendering any binary tree. Since visualizing a BST was a foundational step, they could easily extend that capability to the Red-Black Tree (RBT). As a result, we were able to begin the visualization and animation work before the RBT logic was fully completed, streamlining our project and avoiding potential delays. To ensure smooth and synchronized visualization of the tree's operations, we utilized a queue to manage the sequence of instructions sent to the visualization team. The queue pushes an "Instruction" struct variable to a queue which is kept as a global variable to be accessible by the visualization side. The "Instruction" struct contains the state of the tree before the next rotation so the queue becomes filled with frames of the tree at different stages before and after each rotation and rebalancing. This allows for our base RBT to still perform a worst-case time of O(log N) for all functions while still being able to view the underlying processes occurring pushing things to the visualization side step by step.

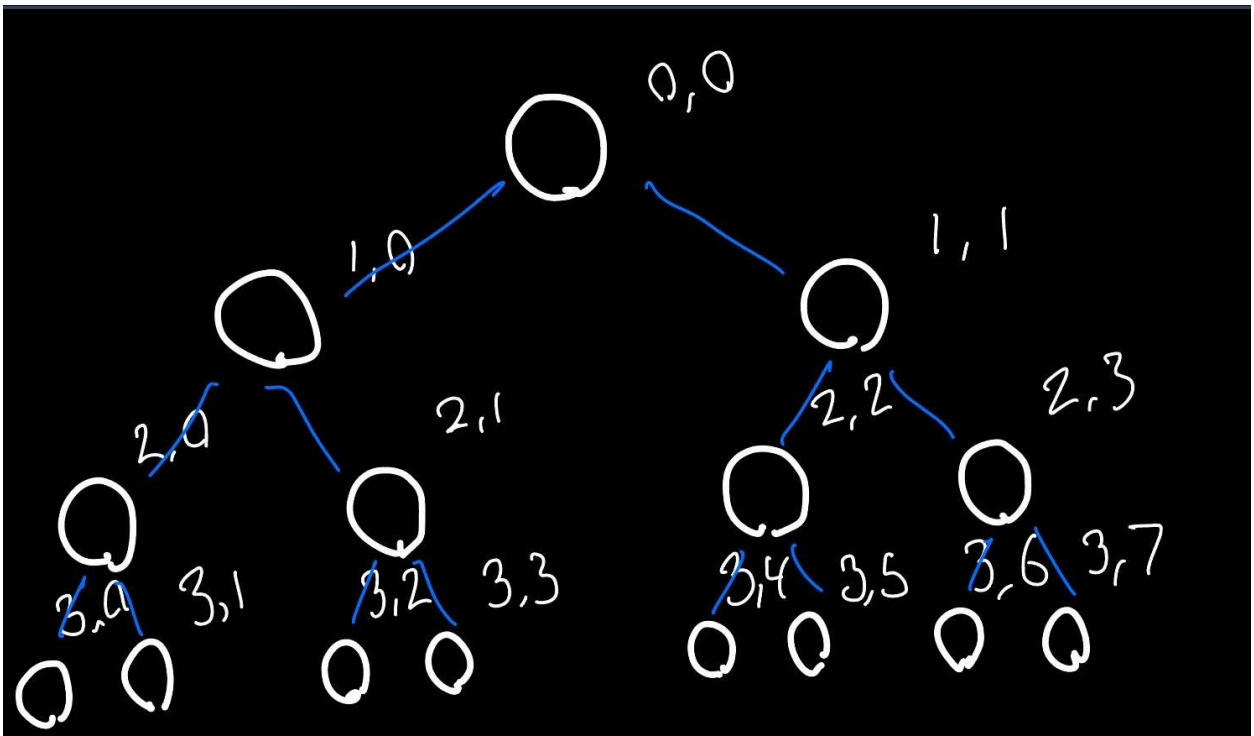
User Interface

Our user interface is purposefully simple. We have one location where you are able to enter in values (only positive integers are accepted) and we have a few buttons to the right of that to keep things fairly simple. Having a simple interface makes this tool usable by anyone. The tree's functions, which users can trigger through buttons, include inserting a node, deleting a node, finding a node, and clearing the tree. Additionally, for convenience, we configured the enter key to submit the number typed in the field to insert into the tree. All of these actions trigger an animation that goes through the steps of the action, highlighting the rotation and visualizing the reorientation of the tree in action.

Challenges

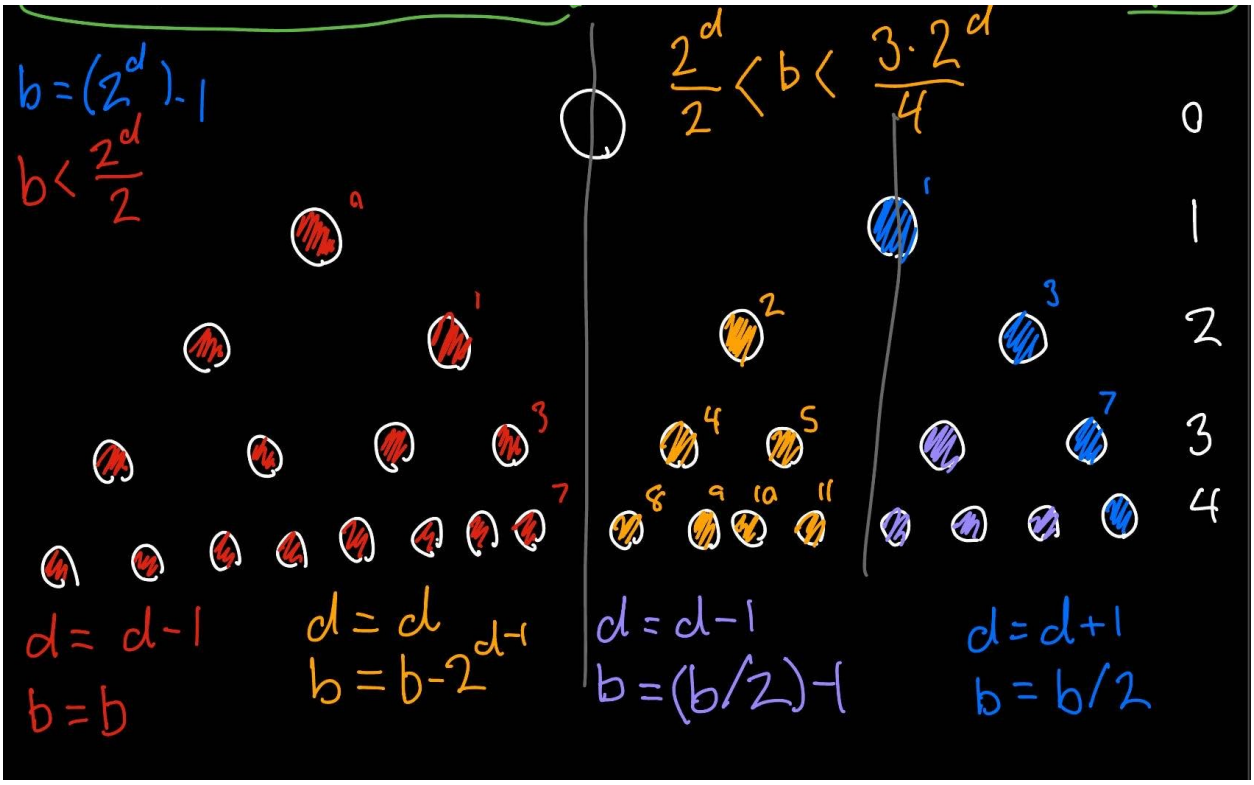
We separated the project into two components, the creation of the tree, and the visualization of the tree. For the RBT logic, the main issue we were running into was the reorientation of the tree after deletion. For both the insertion and deletion of nodes, we encountered numerous edge cases that required careful handling. One particularly challenging scenario arose when deleting the root node. Our rebalancing logic depended on the presence of a parent node; however, since the root has no parent, this logic led to unexpected results or caused the program to crash. As for visualization, the first big issue we ran into was how to display the tree, in

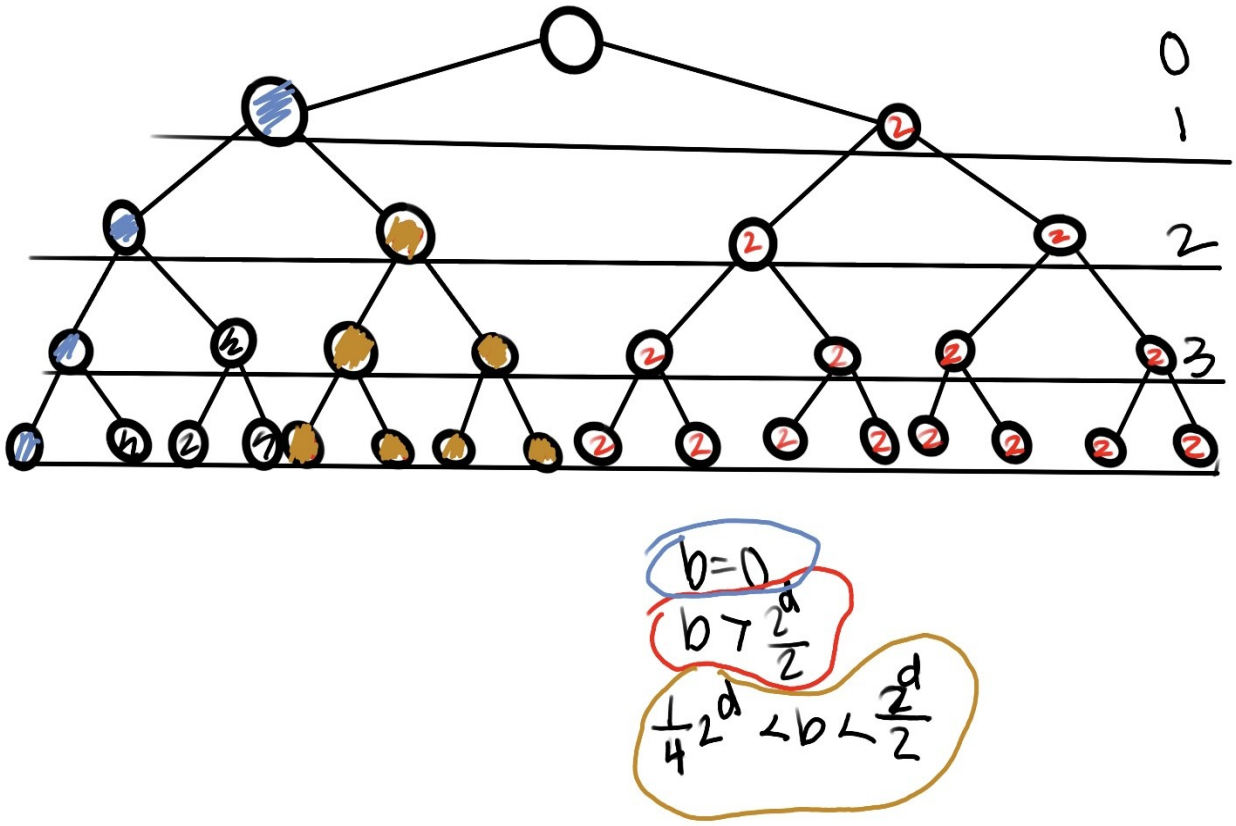
the case that leaf nodes weren’t hugging the left side, so we had to figure out an imagined 2D matrix in which the depth and breadth determined each node's position on the screen. This helped us to realize that given any node’s breadth you can find the path to it from the root by converting that number to binary and assuming that each right turn was a 1, and each left turn was a 0 (Image 1). But by far the largest issue for visualization was breaking down the insertion and removal of nodes into steps that would then be animated sequentially. So we decided whenever we insert a node and do any sort of rebalancing we would push the state of the tree before rebalancing and then animate that tree’s nodes all moving to the next correct location, creating a queue of frames or steps of the tree before reaching the final state after balancing. The problem with this implementation is that on the visual side, we had to derive numerous functions to separate the nodes based on different behaviors when left or right rotating (Images 2 and 3). These functions serve as boundaries to establish various groups of nodes for which the translation in a left or right rotation is the same as other nodes in their respective group. This by far took the most time and caused the most trouble trying to debug, while it was a learning experience for the visual team we would probably choose to do it another way next time.



3,5 R, L, R

0	0	0	0	L	L	L
0	0	1	1	L	L	R
0	1	0	2	L	R	L
0	1	1	3	L	R	R
1	0	0	4	R	L	L
1	0	1	5	R	L	R
1	1	0	6	R	R	L


$b = (2^d) - 1$
$b < \frac{2^d}{2}$
$\frac{2^d}{2} < b < \frac{3 \cdot 2^d}{4}$
0
1
2
3
4
$d = d-1$
$b = b$
$d = d$
$b = b - 2^{d-1}$
$d = d-1$
$b = (b/2) - 1$
$d = d+1$
$b = b/2$

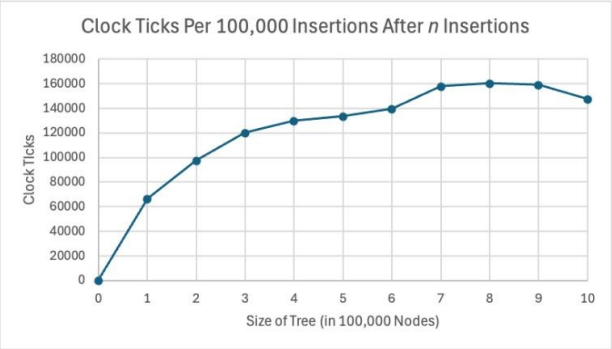


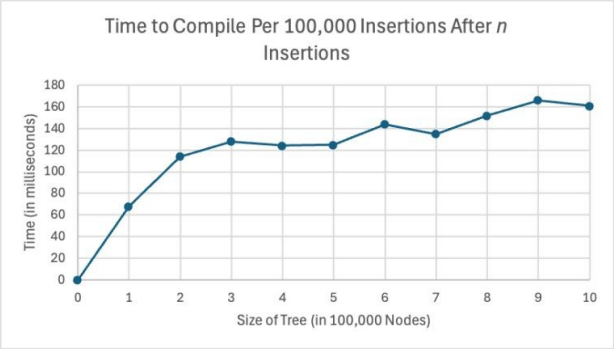
Performance Analysis

As for the asymptotic analysis of our RBT, we performed three tests with different amounts of data. Each table below represents a test with different amounts of data. The following graphs illustrate the asymptotic analysis of a random data set, a data set in ascending order, and a data set in descending order. As you can see, all of these data sets run in O(log n) time. You can tell by the shape of each of the graphs. One of the main differences between these test cases is the rotation and recoloring count.

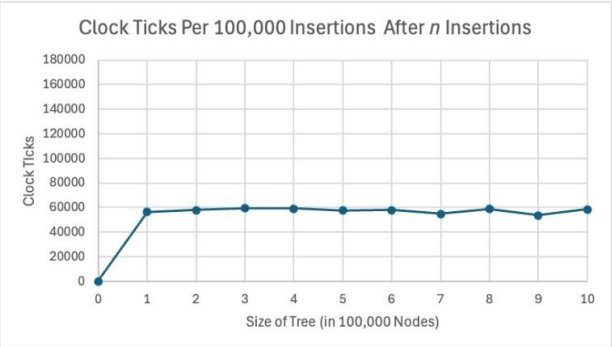
The random data set has the least amount of rotations and the least amount of color changes, but it takes the longest amount of time and the most clock ticks to complete the insertions.

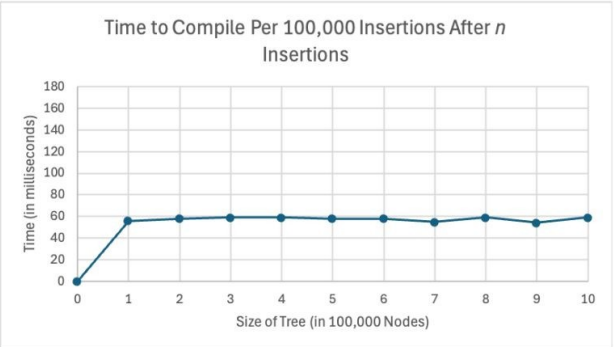
Random Order		
Insertions (in 100,000s)	Clock Ticks	Milliseconds
0	0	0
1	66413	68
2	97315	114
3	120102	128
4	129914	124
5	133384	125
6	139596	144
7	157854	135
8	160186	152
9	159148	166
10	147331	161
Rotations: 588396	Color Changes: 3337895	



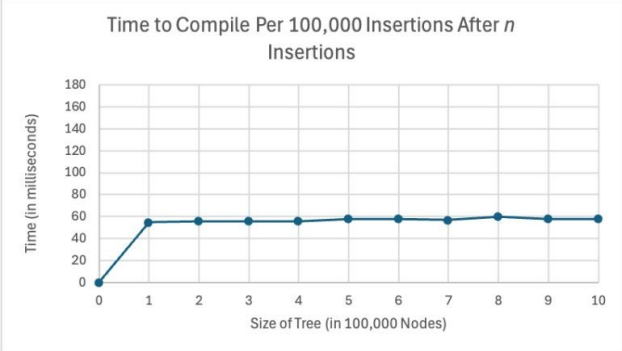


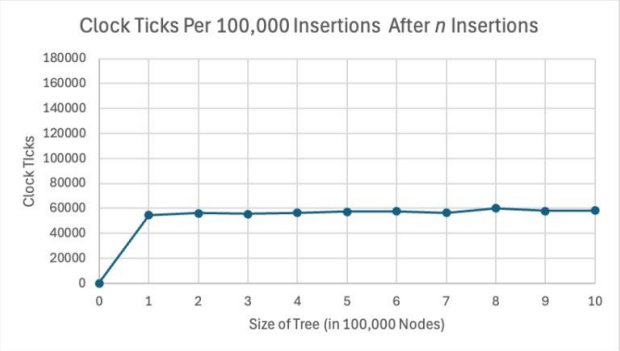
Ascending Order		
Insertions (100000)	Clock Ticks	Milliseconds
0	0	0
1	56353	56
2	57943	58
3	59354	59
4	59157	59
5	57743	58
6	57935	58
7	54854	55
8	58896	59
9	53752	54
10	58595	59
Rotations: 999963	Color Changes: 5999797	





Descending Order		
Insertions (100000)	Clock Ticks	Milliseconds
0	0	0
1	54585	55
2	56175	56
3	55623	56
4	56312	56
5	57507	58
6	57739	58
7	56515	57
8	60140	60
9	58076	58
10	58140	58
Rotations:999963	Color Changes:5999797	





Works Cited

[1] Weiss, Mark Allen. *Data Structures and Algorithm Analysis in C++*. Pearson, 2014.