Lab Report 2: **Analysis of Tree Algorithms**

Akash Santhanakrishnan, Paarth Kadakia, Virendra Jethra

March 19 2023

Contents

List of Figures

1 Introduction

1.1 Overview

The lab focuses on implementing, analyzing, and optimizing self-balancing tree algorithms. The lab will span two weeks and cover:

- A complete implementation of Red-Black Trees (RBTs)
- An analysis of how much better if at all RBTs are over simple BSTs
- Empirically analyse a formally defined tree structure to try and show its height it O(log n) where n is the number of nodes.

1.2 Purpose

This lab report conducts and discusses various RBT and BST tree algorithms experiments. There will be code, graphs, and other visuals to supplement the analysis of these algorithms.

2 Executive Summary

This lab report analyzes the performance of binary search trees (BSTs) and red-black trees (RBTs) through several experiments. The aim of the lab is to provide a complete implementation of RBTs, analyze how much efficient are RBTs are compared to BSTs, and showing that the height of a defined tree structure is O(log n) where n is the number of nodes. The experiments in the lab, include measuring the average height difference between BSTs and RBTs, as well as investigating the impact of sortedness on the average height difference. The results show that the height difference between the two tree algorithms is small, with RBTs being more efficient due to their self-balancing properties. The height difference decreases as the list becomes more randomized. Overall, the lab provides a comprehensive understanding of the performance of BSTs and RBTs. Also it highlights the importance of choosing the appropriate tree algorithm depending on the use cases.

3 Lab Part 1

3.1 Experiment 1

Outline:

- Looking at the average height difference between BST and RBT tree algorithms
- **List Size:** 10000 | **Step:** 1000 | **Runs:** 100

Results:

Figure 1

Conclusion:
As you can see the average height difference between BSTs and RBTs is quite small. This is because the average height of both trees is $O(logn)$. However, the height of RBTs is slightly smaller due to their self-balancing properties, meaning search and insert operations will also be $O(logn)$ whereas for BSTs it will be linear time. An example is if you inserted in the order {2,3,1} the time complexity would be $O(logn)$ but if

you inserted in the order {1,2,3} then the complexity would be $O(n)$. An RBT will always reorganize itself so that you will always maintain the $O(logn)$ so its height will be more balanced. Thus, for use cases where you only need the height of the tree, you may prefer to use BSTs as you will save more memory, but if you want to also use insert and search operations then RBTs are more efficient.

3.2 Experiment 2

Outline:

- Looking at the impact of sortedness on the average height difference of BST and RBT
- **List Size:** 10000 | **Step:** 100 | **Swaps:** 10000 | **Runs:** 100

Graphs:

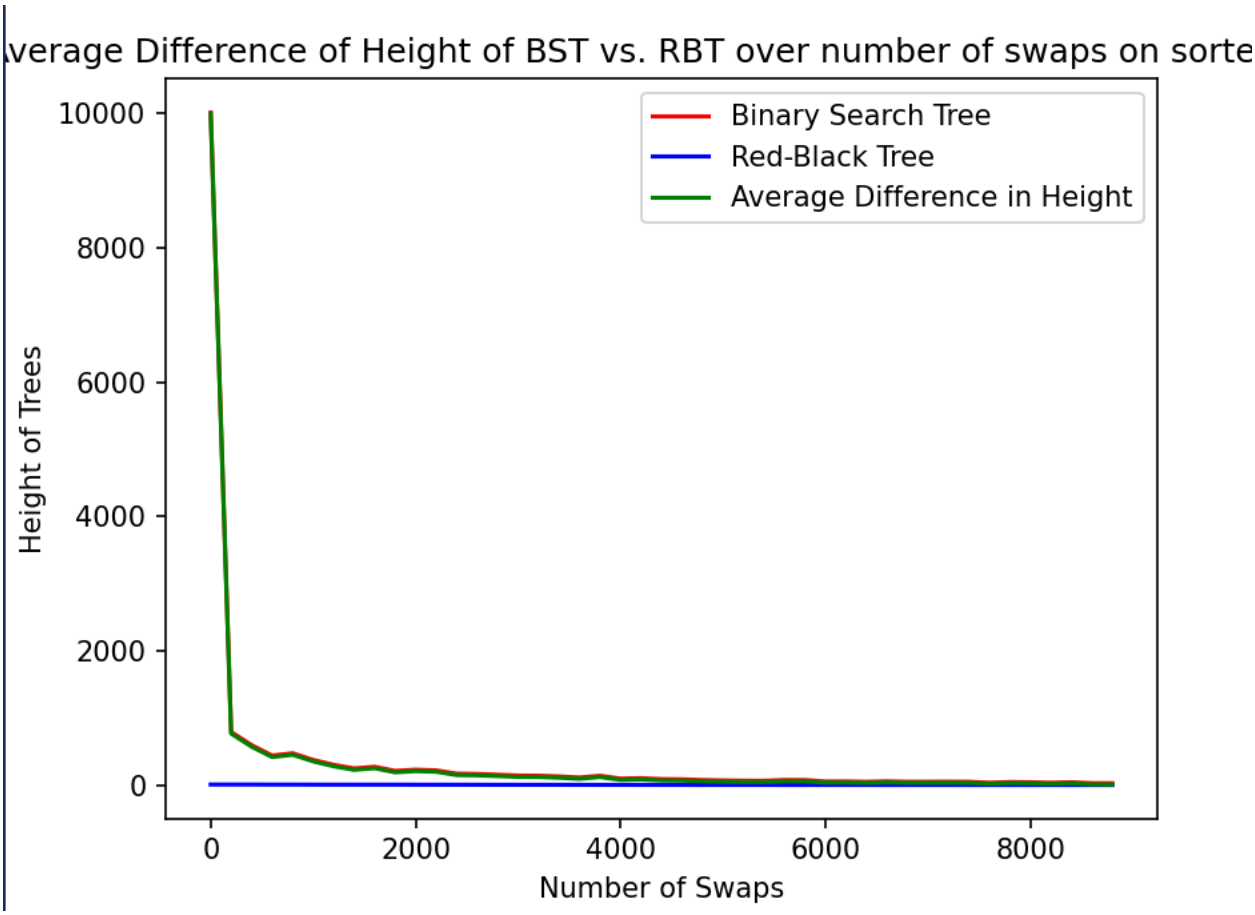

Figure 2

Conclusion:
As you can see the average height difference between BSTs and RBTs is very high at the start and decreases exponentially as the list becomes more randomized. This is because the height of the BST is at its worst case $O(n)$ when the list is sorted and decreases as the list becomes more randomized. The height of the RBT remains $O(logn)$ because of its self-balancing properties, which suggests that RBTs are better suited for handling unsorted data. This is because it is able to maintain balance and limit the growth of the tree height even as the data becomes increasingly unsorted.

4 Lab Part 2

4.1 Experiment 3

Outline:

- Making a height equation based of this custom XC3-tree
- **Degrees:** 25

Graphs:

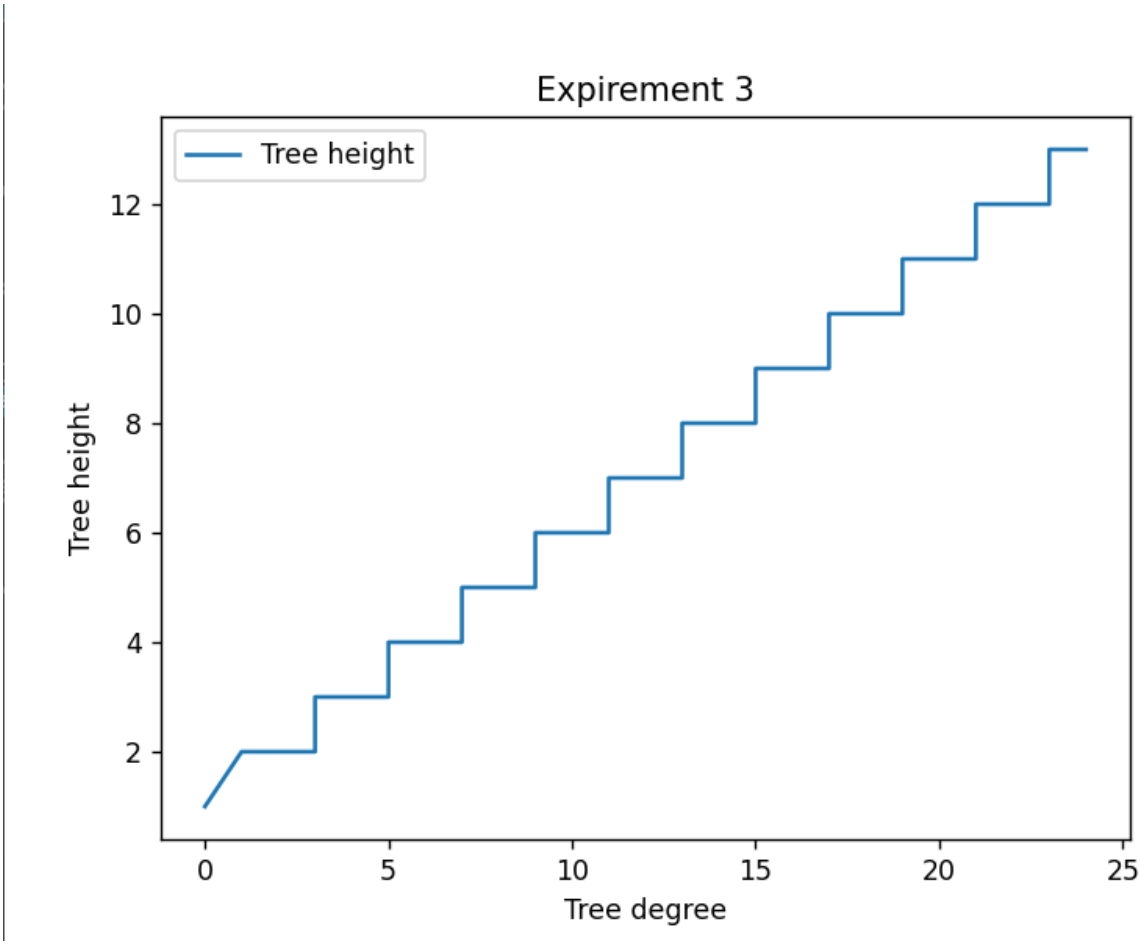

Figure 3

Conclusion:
The tree height equation can be summarized in $h(i) = degree/2$. This can be found simply through looking at the trends within the graph. It is interesting to discuss the fact that at every other degree the tree height increases. If you follow the pattern of the very lowest height leaf nodes they should alternate between having one at the bottom if it is a tree of odd degree number of 3 if it is even. This means that the height of the tree should be increased every odd number degree.

4.2 Experiment 4

Outline:

- Write a node equation based of this custom XC3-tree

- **Degrees:** 25

Graphs:

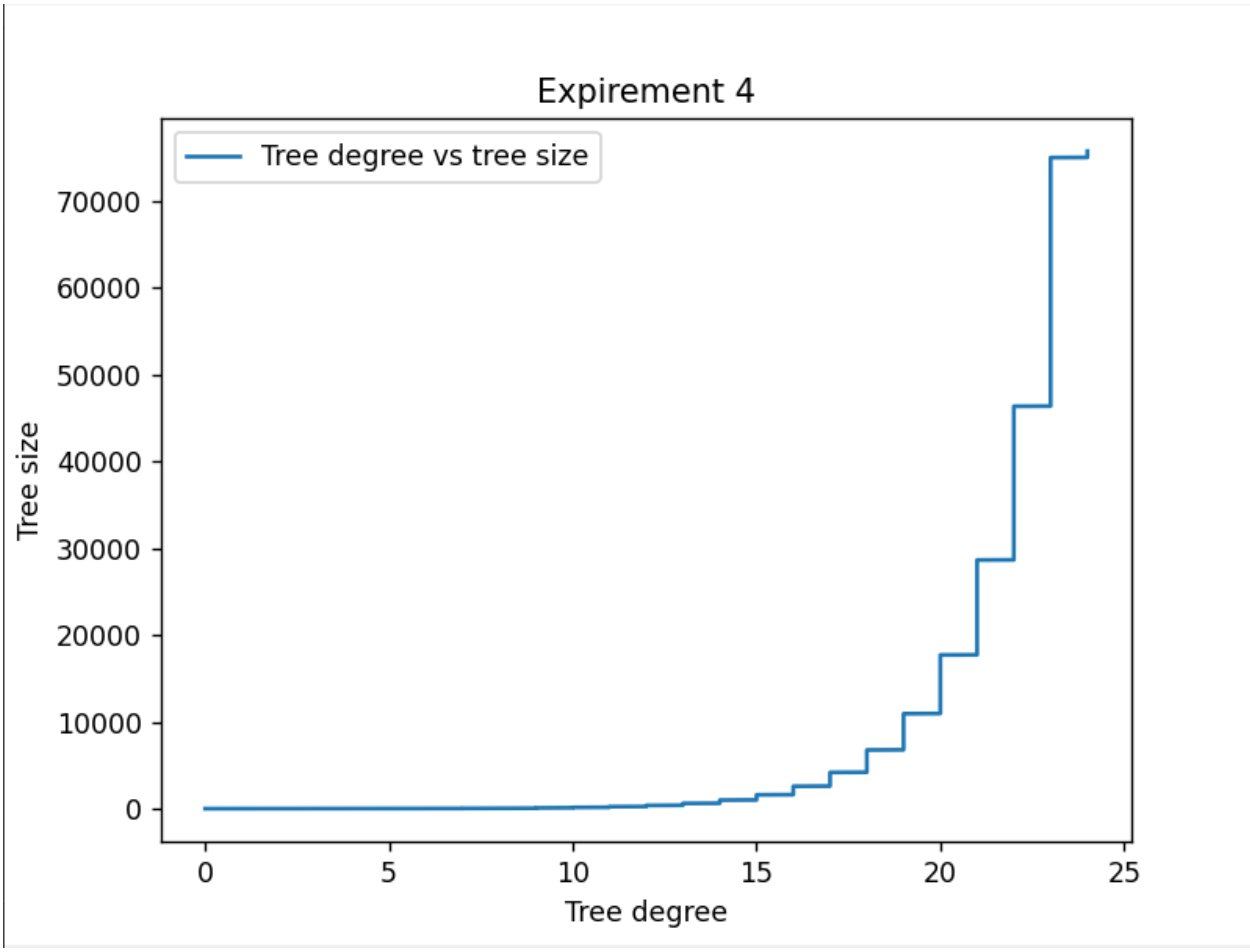

Figure 4

Conclusion:
The node equation looks very similar to e^x based on this graph. In this experiment, we can observe that the number of nodes in an i-degree XC3-Tree equals the sum of the number of nodes in the $(i-1)$th and $(i-2)$th Fibonacci numbers. That is, $nodes(i) = nodes(i-1) + nodes(i-2) + 1$, where $nodes(0) = 1$ and $nodes(1) = 2$. So the pattern is not surprising, as it is just the Fibonacci sequence, where each number is the sum of the two preceding ones. In other words, XC3-Trees exhibit a Fibonacci-like structure, where the number of nodes in an i-degree tree equals the $(i+2)$nd Fibonacci number.

To argue that the height of an XC3-Tree with n nodes is $O(logn)$, we can use the fact that the nth Fibonacci number is approximately equal to $(\phi^n)/\sqrt{5}$, where ϕ is the golden ratio. This implies that the number of nodes in an i-degree XC3-Tree is bounded by $(\phi^{i+2})/\sqrt{5}$ nodes.

Therefore, the total number of nodes in an XC3-Tree of height h is bounded by the sum of the number of nodes in all degree-i XC3-Trees, where i ranges from 0 to h. The following results are obtained by using the upper bound.

5 Appendix

Code for all experiments can be found on this GitHub repository. There are two files one for Part 1 and one for Part 2. Each experiment corresponds to the method number, e.g. Experiment 1 corresponds to the method "experiment1" in the file. Additional sorting algorithms are added underneath the original algorithms to maintain organization and easy navigation.