Runtime Analysis of Treaps and Red-Black Trees:

Capturing Efficacy of Randomized Data Structures

Nicholas Carolan Connor Haugh Max Liu Ashira Mawji

I. Introduction

We strive for rapid data access in computational problem solving, especially with databases, parallel programing, and similar tasks. These applications frequently use binary search trees (BSTs) as a data structure. For operations on a binary search tree to be most efficient, the tree should be balanced such that its max height is logarithmically proportional to its number of nodes. With a balanced BST, we can expect a runtimes of O(lg n) for the *find()*, *insert()*, and *delete()* operations. However, a binary search tree does not guarantee an O(lg n) runtime, as a tree's shape and balance depends on its inputs. Consider when the inputs' keys inserted into the binary search are in strictly increasing or strictly decreasing order, causing the binary search tree to behave like a linked list - with a worse expected runtimes of O(n). Inputs can be unpredictable and sometimes an attacker might deliberately give a worst-case input to greatly increase the runtimes.

Variations of binary search trees that self-balance, independent of input, are therefore advantageous. A novel solution to this challenge is the Treap. Treaps combine BST rules, Heap properties, and randomization into nodes' keys and priorities and thereby maintain a balanced data structure. This experiment compares Treap performance to a similar but discrete solution, the Red-Black Tree. Using a wide distribution of requests, we attempt to gauge the relative effectiveness of the two solutions, which both claim O(lg n) runtimes. By juxtaposing the results of this experiment, we can begin to understand the value of randomized data structures in improving performance with simple solutions.

II. Experimental Setup

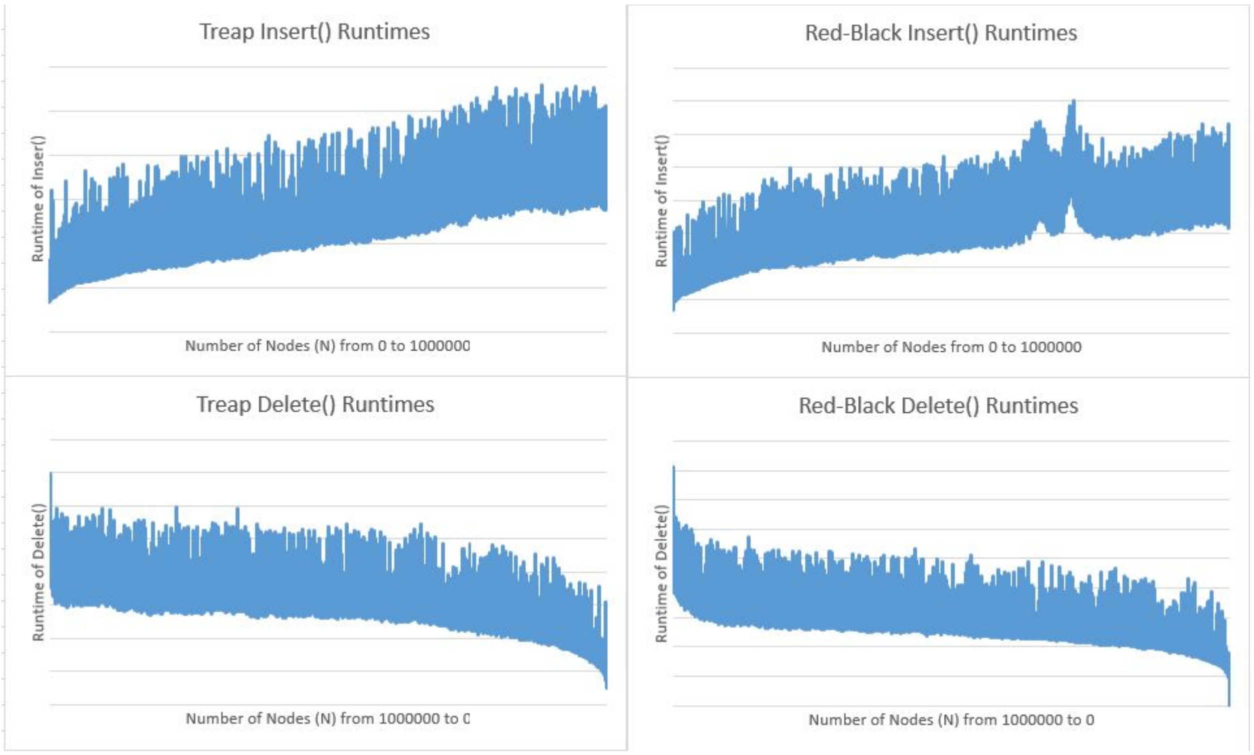
In this project, the priorities are assumed to be continuous random variables drawn from a uniform distribution. Therefore, all priorities are distinct and equally likely to be chosen. Once a priority is randomly generated and paired with a key, it is fixed and will not change under any Treap operations.

This experiment compares the average runtimes of two standard operations of Treaps and Red-Black Trees, *insert()* and *remove()*. Initially, 1,000,000 distinct elements are generated from a uniform random distribution and inserted into the relevant data structure. All 1,000,000 elements are then deleted in a random order from each data structure. This process - 1 ,000,000 insertions followed by 1,000,000 deletions - is repeated 100 times. Meanwhile our program records the system runtime for each individual *insert()* and *delete()* operation, then averages the runtime of each operation at each number of nodes over the 100 trials. Thereby we compare the average runtime of each operation at each of the 1,000,000 node quantities in each data structure. Over this large number of operations on random data, we expect that comparing the operation runtimes will reveal that Treaps have a performance advantage over Red-Black Trees. Additionally, plotting the runtimes for each data structure with a large number of elements will indicate how runtimes behaves asymptotically.

III. Results & Discussion

Graph 1 displays the relationship between the number of nodes in the data structure and the runtime of the relevant operation, either *Insert()* or *Delete()*. Approximately 400 data points were removed in each graph in order to smooth analysis of the function; these data points were designated outliers if they were more than 500 nanoseconds away from both their neighbors. As is visible in the graph, both the Treap *Insert()* and Red-Black *Insert()* functions mirror have lg n shapes, with runtime increasing as the number of elements in the data structure at the time increases. The Treap *Delete()* and Red-Black *Delete()* functions also have lg n shapes but appear to be flipped across the x=500,000 axis because their runtimes decrease as the number of elements in the data structure decrease, and their axis begins at 1,000,000 nodes and ends at 0 nodes.

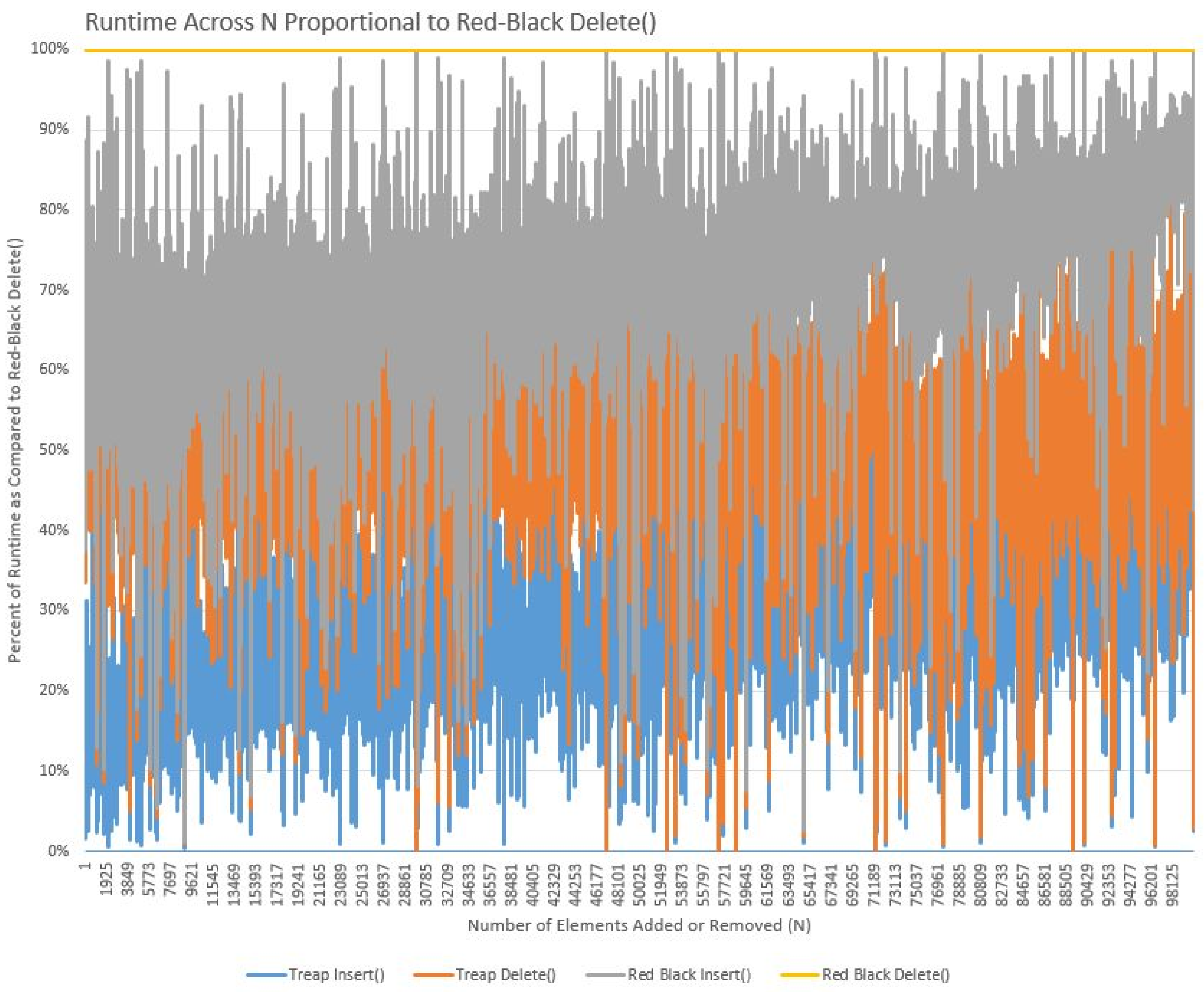
Graph 1: Runtimes of Treap *Insert()*, Treap *Delete()*, Red-Black *Insert()*, and Red-Black *Delete()*



Both Red-Black Trees and Treaps predict runtimes of O(lg n) under asymptotic analysis, although Red-Black Trees have a strict upper bound of O(lg n) runtimes while Treaps merely have an O(lg n) runtime with high probability. In Graph 2 below we compare the runtime of each operation to the consistently slowest of the four, Red-Black *delete()*. Across the experimental results, the

Red-Black *delete()* operation consistently produced the longest runtimes. Consequently, modeling the results in proportion to this operation shows the relative effectiveness of the Treap. Treap *delete()* can be viewed as taking 50% the time of its Red-Black counterpart, while Treap *insert()* takes roughly 50% less time than Red-Black *insert()*.

Graph 2: Operation Runtimes vs. Red-Black *Delete()* Runtime



IV. Conclusion

The two data structures compared in this experiment attempt to ensure balanced binary search trees in the quest for rapid insertions and deletions in large data sets. Through randomization, the Treap provides a somewhat less complex and less predictable alternative to the Red-Black Tree. Treaps do not require specific coloring for each node or a complex set of rotations and recoloring to preserve their structural properties. Additionally, the random generation of priorities means that the final structure of a treap cannot be predicted from a set of key inputs. This prevents a malicious actor from providing a worst-case set of insertions to maximize runtime.

The runtime analysis of these data structures indicates that Treaps are a more time-efficient solution than Red-Black Trees across a large range of data sizes. Although both structures guarantee average runtimes of O(lg n), Treaps empirically performed better in our experiment for both *insert()* and *delete()*. Red-Black Trees performed more slowly across all data sizes - even operations with a small number of elements showed meaningful differences in efficiency. In utilizing tree structures, whether for parallel computing, database management, or page-tree implementations of virtual memory systems, Treaps may very well be the preferred alternative to Red-Black Trees. Not only do they provide better runtime than Red-Black Trees, but they are also implemented using a simpler and more intuitive balancing mechanism.

Our analysis is limited by exploring the behavior of repeated references. Effective data structures must pay heed to usage patterns, and many usages of trees, such as searches, may require the lookup of specific nodes multiple times. The implementation of Treaps in this experiment has no way to make data more accessible (i.e. closer to the root and quicker to access). In future studies of Treaps, we would be interested in manipulating a node's priority to optimize search time for more frequently called keys.

Furthermore, we would like to compare spatial efficiency of the Treap relative to that of the Red-Black Tree. In the testing process, running our *insert()* and *delete()* experiments on the Red-Black Tree produced heap overflow errors before reaching 100 iterations whereas space was not a problem for corresponding Treap experiments. This suggests that the additional information required to balance a Red-Black Tree, like color, requires significantly more space than merely the Treap's priority values. This is more noteworthy given that in theory the Red-Black Tree library used to create the Red-Black Tree was subject to rigorous open-source scrutiny and is therefore spatially optimized, while we coded our Treap ourselves. Consequently, we might be interested in analyzing the space requirements of a Treap compared to a Red-Black Tree or standard BST.