Implementing Red/Black Trees

Carl Ehrett, Stephen Peele

November 29, 2016

1 Why Red/Black Trees?

In the course of using a binary search tree (BST) for data storage, it might be the case that the tree becomes highly imbalanced. That is, with respect to the parent node, one subtree becomes much deeper than the other. For small-scale applications, this might not be of much concern; however, for a large-scale application, this imbalance can greatly deteriorate the efficiency of operations on the tree and hence the application. Since the operations on a binary search tree can be of $\mathcal{O}(n)$ complexity in the worst case, it would be wise to consider an implementation which keeps the tree balanced so as to control its depth which in turn improves the performance of the application.

This leads one to red/black trees (RBTs). an RBT is a BST with the additional properties:

- Every node is “colored” either red or black.
- Every red node that is not a leaf has only black children, i.e. a non-leaf red node may only have black children.
- Every path from the root to a leaf node contains an indentical number of black nodes.
- The root node is black.

Interestingly, these additional properties constrain the RBT to “balance” itself after an insertion or deletion. This gives the very nice property that the depth is controlled to about $2\log n$ on average for an RBT with n random elements [4]. Thus, for an RBT with 500 random entries, the depth is about twelve (on average)! Since the complexity of operations on a BST are proportional to its depth, the advantage of the RBT is that all operations can be performed in logarithmic time [4]. Thus, for an RBT with n elements, the tree has a worst-case depth of $\mathcal{O}(\log n)$, which is clearly improved from the worst-case BST, i.e. $\mathcal{O}(n)$. As a real instance of the use of RBTs, we note that RBTs are used in the Linux kernel [3].

2 Implementation

We implement the RBT datatype in Python [7]. Our particular implementation of the red/black tree is that of a left-leaning red/black tree (LLRBT) which is a variant of the RBT proposed and implemented by Sedgewick [4]. As stated in his paper, the LLRBT is simpler to implement and achieves the same goals of an RBT. We draw upon his implementation as our inspiration.

The implementation of a BST is fairly straightforward. Hence, our main difficulty encountered was ensuring that the RBT properties are restored after an insert or delete operation. This was the most complicated aspect of the implementation.

3 Time Comparisons

To better understand the achievements of our RBT implementation, we tested it versus that of a typical BST using the Python `timeit` module for some of the standard operations. The nature of these tests are described in the code. We tested on the SageMathCloud architecture for $n = 500$ items in the RBT/BST [7]. The results are outlined in the table below. Note the clearly faster result which is highlighted in blue for emphasis.

Operation	Structure	Average	Worst
Insert/Fill	RBT	13.6 ms	6.78 ms
	BST	4.65 ms	47 ms
Lookup	RBT	1.29 ms	1.21 ms
	BST	1.49 ms	42.8 ms
Fill/Empty	RBT	20.2 ms	6.81 ms
	BST	4.11 ms	46.3 ms
Traversal	RBT	-	216 μs
	BST	-	239 μs

As anticipated from our research on the topic, the RBT performs better than the BST in each of the worst-case scenarios. Note that for the average case for insert and fill/empty, the RBT is slower than the BST. This makes sense given the additional computational overhead in the RBT; however, for the worst case, the RBT appears to be far and away much more efficient. For lookup, the RBT and BST are nearly identical for the average case, but the BST is far slower for the worst case, which is expected. We note that lookup in the RBT is nearly identical for the average and worst case. Finally, traversing each of the trees is fast with nearly identical times.

4 RBT Complexity

In average cases, the complexity of lookup, insert, and delete in a binary search tree is $\mathcal{O}(\log n)$. This is because in the average case, the height of a binary search tree is proportional to $\log n$; however, in the worst case—specifically, with sorted input—a binary search tree can be extremely unbalanced, so that the height of the tree is n. Thus in the worst case, the complexity of lookup, insert, and delete in a binary search tree is $\mathcal{O}(n)$.

A red/black tree enforces that the tree is close to balanced. The result is that the complexity, even in the worst case, remains at $\mathcal{O}(\log n)$ for lookup, insert, and delete. A comparison of time required to insert n sorted entries, for various n, appears in Figure 1 on the next page.

5 Conclusion

As we have shown, the RBT is an variation of a BST which constrains the tree to maintain some sense of balance. For worst-case behavior, the RBT is more efficient than the standard BST in all of its operations; however, we note the additional computational overhead for the RBT which may allow the BST to be slightly more efficient in average-case behavior. For small applications or applications in which balance is not of much concern, not much is gained from the RBT. Indeed, it may actually slow things; however, an application in which balance is an issue, the use of an RBT (or some other self-balancing BST) would almost assuredly be required as any chance of attaining (or maintaining) speed and efficiency would be nullified by the occurrence of a highly unbalanced BST.

Finally, we note that although our implementation is fast enough for our purposes in Python, the nature of Python certainly slows down this implementation. We surmise that the gains would be even more substantial in a language such as C or C++.

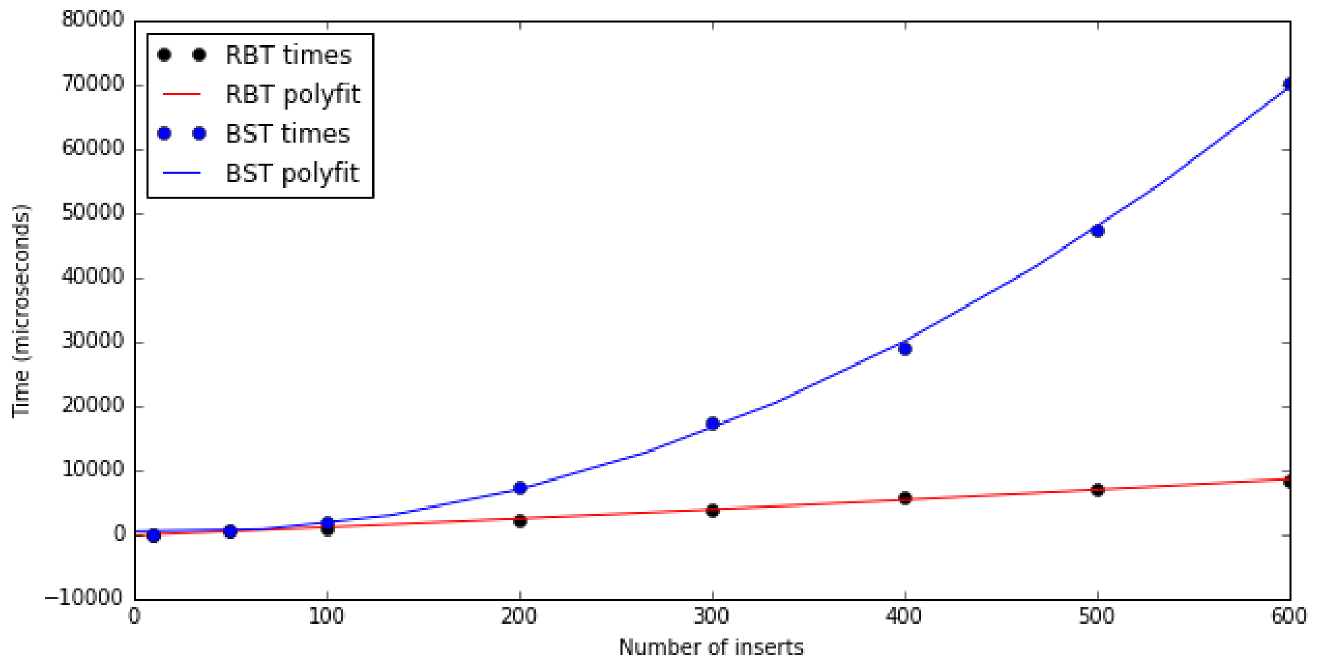

Figure 1: Time required to insert n sorted entries in microseconds (μs). Note the clearly better performance of the RBT as n grows.

6 References

[1] Dr. Timo Heister (lecture notes, homework code, suggestions, insight, etc.)

[2] Red/Black Tree Visualization. https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

[3] *Trees II: red-black trees.* https://lwn.net/Articles/184495/

[4] Sedgewick, 2008. *Left-leaning Red-Black Trees.* https://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf

[5] *Red-black tree.* https://en.wikipedia.org/wiki/Red-black_tree

[6] *Left-leaning red-black tree.* https://en.wikipedia.org/wiki/Left-leaning_red-black_tree

[7] Code submitted as an iPython Notebook created on the SageMathCloud architecture.