Name: Timothy Holland
Random seed: 73

Splay Tree Experimentation Report

The goal of this report is to experiment with two implementations of splay trees. First, a "naïve" implementation which splays using only single consecutive roations, rotating the node to be splayed with its parent over and over until it reaches the root. Second, the "standard" implementation, which uses the more complicated double roations (Zig-Zig, and Zig-Zag), to move the node to be splayed to the root through rotations with and between its parent and grandparent.

At the outset, the pressing question is why is there a need for the standard implementation in the first place? I will consider this question in a practical sense, discussing the experimental results first, and utilising theory to support my arguments. Ultimately, I will show that the naïve implementation reduces to the worst cast of binary search trees without a balancing technique ($O(n)$ with an additional constant overhead for the splay operation). Whereas, the standard implementation avoids this worst case in the amortised sense, by being able to construct a 'balanced' tree even in this worst case scenario, achieving $O((m+n)\log(n))$, where m is number of splay operations on a tree with n nodes, for the splaying of a node (which becomes essentially logarithmic the more operations performed). Moreover, I will make two further arguments. First, that the despite the better amortised cost of the standard implementation, it is ultimately a constant factor worse in performance than the naïve implemnetation when we consider average case complexity (Big Θ). Second, I will hypothesise that despite the naïve implemenation's worst case, if any randomness is introduced into the structure, then it can recover over to a logarithmic bound over its lifecycle.

Sequential Test

In the sequential test, n elements are inserted into the tree sequentially, and are then repeatedly found in sequential order. Again, for non balancing trees, this would amount to a reduction to a list, with insert and find operations both taking $O(n)$ in the given size of the tree. The following two graphs (Fig.1 and Fig.2) illustrate the difference in performance between implementations:

Name: Timothy Holland
Random seed: 73

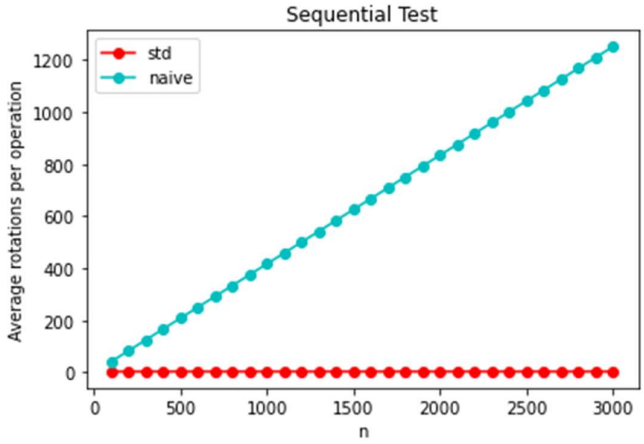

Fig.1: Coparison over sequential test

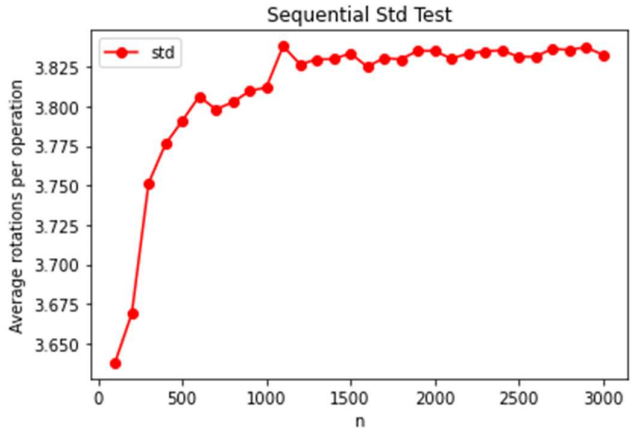

Figure 2: Standard implementation

As can be seen in the above two graphs, the naïve implementation reduces to linear time for insertion and find operations, whereas the standard implementation becomes logarithmic in both. Why this performance difference? Or rather, why does the naïve implementation reduce to the worst case of a non-balancing binary search tree?

The theorem of amortised cost of splaying a given node states that it is at most $3 \cdot (r'(x) - r(x)) + 1$, where r(x) is the rank of the node before the operation and r'(x) the rank after it. This is proved by considering the amortised cost of the full splay as the sum of amortised costs of each individual step. Thus, to consider the difference in implementations, we can consult cost analysis to illustrate why double rotations in the standard implementation is necessary to amortise the worst-case complexity to logarithmic time.

Let's first consider the naïve case consisting only of single rotations. For a single Zig rotation, the real cost is 1, swapping the node x with its parent y, which thus also changes the ranks of each:

$$A = 1 + r'(x) + r'(y) - r(x) - r(y)$$

By inclusion of the subtrees we have $r'(y) \leq r'(x)$ and $r(y) \geq r(x)$, thus:

$$A \leq 1 + 2r'(x) - 2r(x) \leq 1 + 3r'(x) - 3r(x)$$

We must now consider the cost of this single splay operation after a find or insert, which consists of m operations over n nodes:

$$A \leq \sum_{i=1}^{m} (3r_i(x) - 3_{r-1}(x) + 1)$$

In this equation, the sum telescopes over $3r_i(x)$ and $3r_{i-1}(x)$ but results in an extra cost of m due to the real cost of a single splay:

$$A \leq 3r_i(x) - 3_{r-1}(x) + m$$

Thus, the amoritsed cost of the naïve implementation becomes bounded to m splay operations, despite the cost of ranks being logarithmic. If we consider the worst case sequential scenario, this m is equivalent to n, as there is a splay for each node in the data structure. Thus for find, and insert, there is a worst cast cost of $O(m) = O(n)$ for first finding the given node, which is then repeated upon splaying, multiplying this worst cast by a constant of 2.

Alterntaively, the standard implementation avoids this by maintaining the real cost of a Zig operation across a sequence of operations as remaining constant, and allowing the telescoping of both Zig-Zag and Zig-Zig operations across the summation of operations. The theorem states that for both operations, the amortised cost is:

$$A \leq 3r'(x) - 3r(x)$$

Thus, we can consider the cost of a sequence of m operations as being the sum of this cost for m steps, with an additional constant cost of 1 for a single zig if necessary:

$$A \leq \sum_{i=1}^{m} \big(3r'(x) - 3r(x)\big) + 1$$

Again, the sum telescopes, but the zig operation remains constant, thus allowing us to maintain our initial theorem, as the complexity of ranks in the worst case remain logarithmic.

Name: Timothy Holland
Random seed: 73

Random Test

We have shown that the standard implementation is necessary in order to maintain the amortised cost of all operations as being logarithmic in the worst case. However, we may wonder how this figures over a randomised, that is, normal distribution of data. The random test inserts n elements in random order and then finds 5n random elements from the given structure. The results can be visualised in the following figure (figure 3):

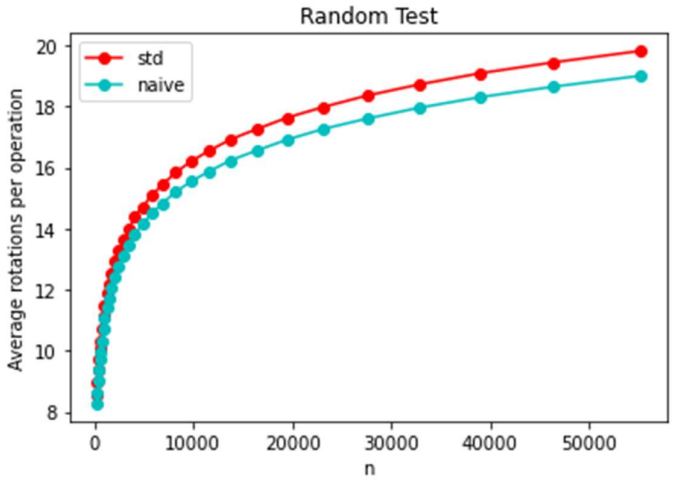

Figure 3: Comparison over random test

Notably, two questions arise. First, why does the naïve implementation become logarithmic when operations are randomised? Second, why does the naïve implementation perform better by some constant factor?

To answer the first question, both implementations result in logarithmic bounding, which we can assume from the random insertion and find, as being distributed normally, and thus giving us an approximation of $\Theta(\log(n))$. Why is this the case? Let's consider our amortised cost analysis over m operations again:

$$A \leq 3r_i(x) - 3_{r-1}(x) + m$$

In the sequential test, m bloats to n, thus resulting in linear worst case complexity. However, we know in the Big Θ sense, random insertion and find results in a roughly balanced tree, with depth logarithmic in the size of n. Consequently, in this case, the cost of the zig operatiosn doesn't linearise with m, but reduces to another logarithmic factor.

To answer the second question, I can only hypothesise as to the answer. The essential difference between single rotations and double rotations, is that grandparents are incorporated in the later. Whilst this enables the construction of more balanced trees in the worst case, I contend that this

operation actually introduces slightly more unbalanced trees in the Big Θ sense, thus resulting in the constant factor difference between these two implementations. Indeed, if we consider the change in subtrees, in each Zig-Zig and Zig-Zag cases, there is potential for subtrees to move two levels (up or down), whereas only one level changes across the naïve implementation. Thus, it may be that the double rotations actually bring more instability across splays which ultimately average out over many operations, but which result in a constant factor difference to the more 'stable' naïve implementation.

Subset Test

Lastly, in the subset experiment, a sequence of n elements is inserted, which contains arithmetic progressions interspersed with random elements, thus fixturing a combination of the above two tests. Then a small subset of these elements are accessed in random order, with progressive cardinalities of 10, 100, and 1000 tried over different iterations. The overall results can be visualised here, comparing the size of the tree with the average rotations per find (figure 4):

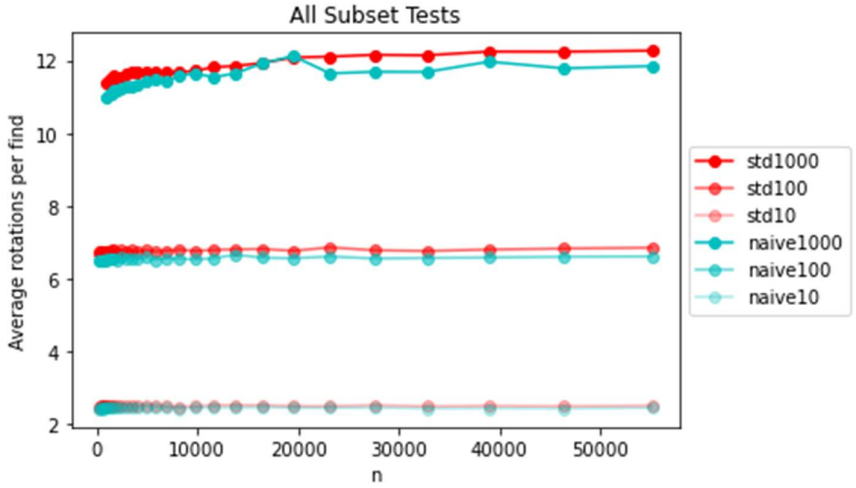

Figure 4: Comparison over subsets

Notably, every operation appears to be logarithmically bounded in n, with the performance of the naïve implementation beating the standard implementation by some constant factor. Overall, as a measure of randomness exists within the insertion of elements in the tree and find operations, these results appear consistent with the above random experiment. Due to scale, a closer look at each subset is required in order to form any greater interpretations from the data, these can be visualised as follows (figures 5, 6, 7):

Name: Timothy Holland
Random seed: 73

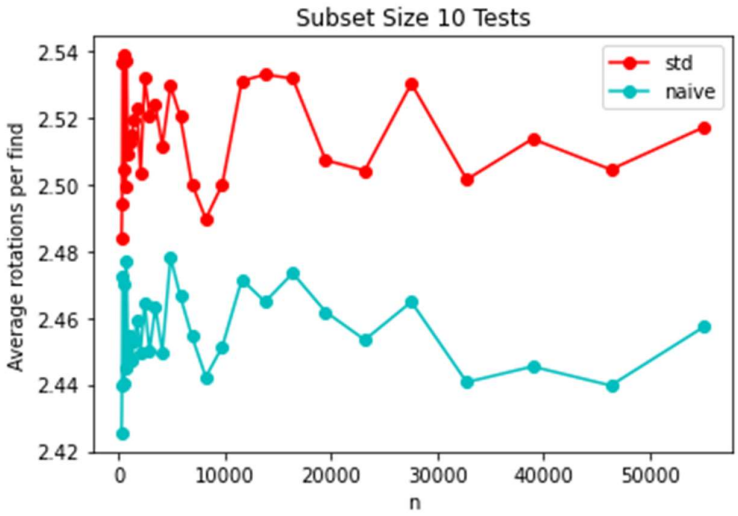

Figure 5: Comparison over cardinality 10

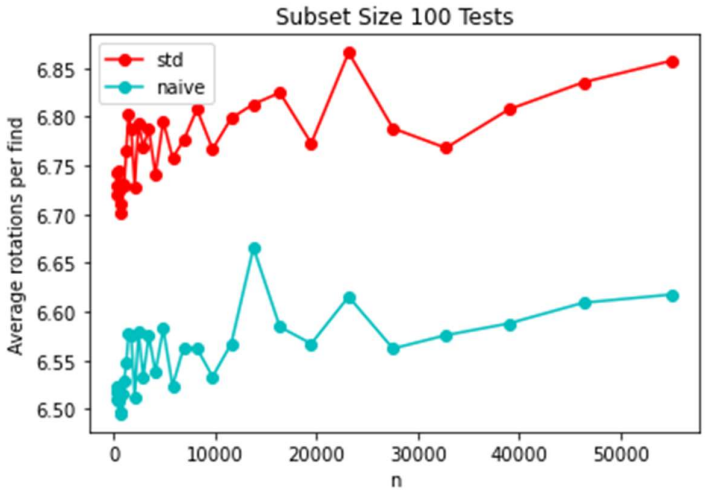

Figure 6: Comparison over cardinality 100

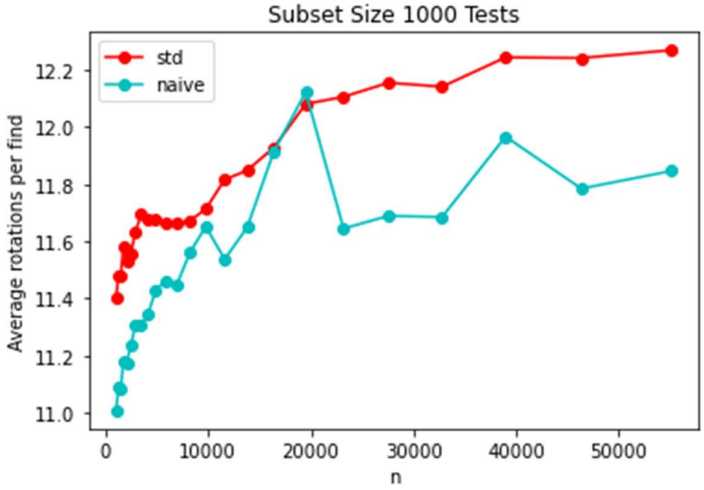

Figure 7: Comparison over cardinality 1000

Again, considering the smaller subsets of size 10 and 100 (figures 6 and 7), the distributions of rotations per find as n increases remains strikingly similar across implementations (with the naïve still greater by some constant factor). The most notable observation comes when we consider subsets of 100 (figure 7). Despite initial results consistent with the above experiments, there is one point where the naïve implementation actually degrades past the standard implementation, but then flattens out to its expected logarithmic bound. We can hypothesise as to the cause of this degradation by considering the first two experiments in tandem. As stated, the subset experiment contains arithmetic progressions interspersed with random elements. Thus we can attribute the performance degradation of the naïve implementation to the dominance of these arithmetic progressions in the tree. Indeed, the average rotations per find appear to be increasing linearly in the size of n, before eventually returning to its logarithmic bound. The question therefore becomes, why doesn't the subset experiment reduce entirely to linearity? Well, by incorporating random elements, the tree doesn't reduce entirely to a list upon creation, and can form a more balanced structure across its lifetime. Thus, the subset experiment reveals that whilst the sequential test isn't affected by sequential progressions, high cardinalities can effect the naïve implementation, making it linear in n initially, before returning to a logarithmic bound. For higher cardinalities, I hypothesise that the performance degradation would be more drastic, with an increasingly undeseriable time to recover.