# Assignment 3: splay\_experiment

Name: Romita Pandurang Pawar

The experiment is based on the previous assignment splay operation and this splay\_experiment runs successfully based on splay operation. The seed used for this experiment was 40 (last 2 digits of student id).

1. **Sequential test:**

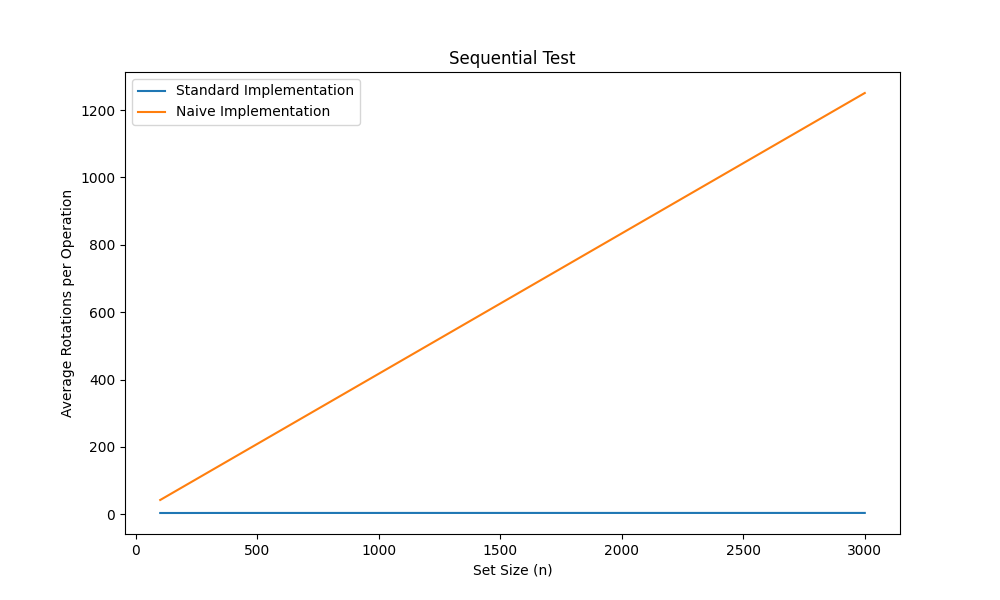


Fig 1: Sequential Graph for std and naive

By looking at the results of sequential graph we can say that there is a huge difference between the number of rotations executed by the standard and naïve implementations.

For both std and naive implementation, the number of rotations and the complexity of adding n elements are in O(n), or constant per element. Whenever this happens, a linear list is produced. Each implementation will utilize this resulting form after completing either the add or search sequence.

Standard implementation: The number of rotations executed as well as for finding the feature, for the standard implementations, here is constant, because complexity is amortized O(α(n)); where; α is the inverse Ackermann Function. This complexity arises due to the self-adjusting nature of splay trees, which reorganize themselves dynamically during operations like search, insertion, and deletion. This analysis indicates that when representing the splay tree as a deque, one should remove the edge points.

Naïve implementation: Over here, the number of rotations is linear. This is because the tree always takes the form of two head-connected lists during the search, from which we remove the leaves of one, and add them to the head of the other. Then, we can achieve a very accurate linear approximation of the dependence of the average number of rotations per field size by a straightforward mathematical change. Therefore, in a naïve implementation, the complexity for determining an element (and the number of rotations) is in O(n). This is because, without the self-adjusting behaviour, the worst-case complexity of individual operations (search, insertion, deletion) in a naive splay tree can be O(n).

1. **Random test**:

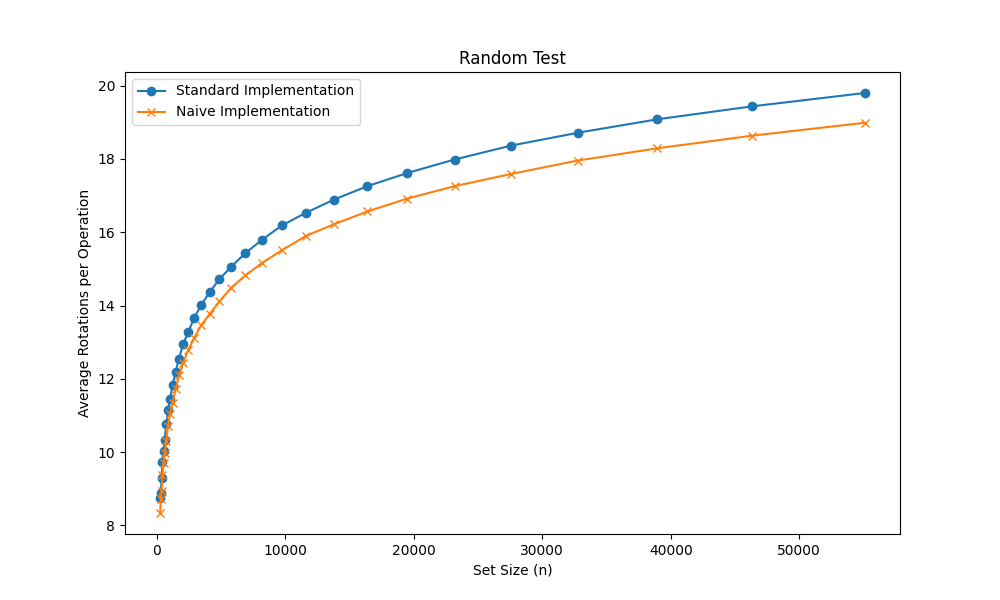


Fig 2: Random Graph for std and naive

Over here in the Random test, the standard and the naïve both implementation curves appear to be logarithmic, as the set size grows, the average number of operations per operation increases slowly and due to the amortized time complexity of operations in splay tree such as search, insertion, and deletion is O(log n), where n is the number of nodes in the tree. This means that, on average, these operations take logarithmic time with respect to the size of the tree. Also, both the implementations gained similar number of rotations. This is because as the number of elements (**set size**) increases, the average time per operation remains relatively low. Splay trees rearrange the tree structure after each access, bringing the accessed node closer to the root. This self-adjustment property ensures that frequently accessed nodes are kept near the root, resulting in faster access times on average. In a random test, we consider a sequence of operations (searches, insertions, deletions) performed on the splay tree. While individual operations may have varying time complexities (some may take O(n) time), the amortized complexity over the entire sequence remains O(α(n)), where; α represents the inverse Ackermann Function.

For Standard: The random test operation takes advantage of the self-adjustment property by splaying the accessed nodes during the operation. This means that as the set size increases, the amortized cost of accessing a random element remains relatively low, as the frequently accessed nodes are kept closer to the root.

For Naïve: This implementation does not exploit the self-adjusting nature of splay trees. It simply traverses the tree from the root to find the random element, resulting in a higher average cost as the set size grows larger. The naive implementation does not benefit from the restructuring of the tree, leading to a higher number of operations required on average.

By comparing, both the implementations, the naïve implementation performs better than the standard one. This is evident throughout the range of set sizes from the lower curve of the naive implementation in comparison to the higher curve of the standard implementation. The reason is naïve splay trees tend to self-balance and the standard implementation leverages the self-adjusting properties of splay trees for efficient random-access operations.

**UPDATE:**

By comparing, both the implementations, the naïve implementation performs better than the standard one. This is evident throughout the range of set sizes from the lower curve of the naive implementation in comparison to the higher curve of the standard implementation. The reason is naïve splay trees tend to self-balance and the standard implementation leverages the self-adjusting properties of splay trees for efficient random-access operations.

In the previous test’s implementation of the random test operation in a splay tree takes advantage of the self-adjusting properties of the data structure. After each access, the splay tree restructures itself by performing rotations to bring the accessed node closer to the root. This property ensures that frequently accessed nodes are kept near the root, resulting in faster access times on average. Whereas, the naive implementation does not exploit that advantages and is likely to have a higher average cost, especially as the set size increases.

{Theorem from the lectures; page 25}

Theorem:A sequence of m operations Find, Insert, and Delete on an initially empty Splay tree takes O(m log n) time, where n is the maximum number of nodes in the tree during the sequence.

1. **Subset test**:

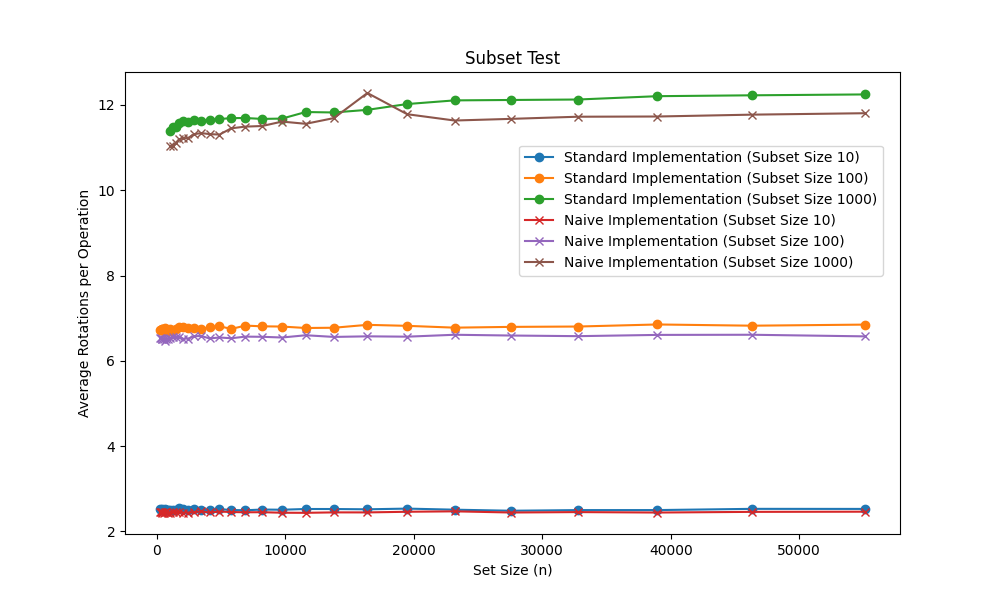


Fig 3: Subset Graph for std and naïve

The subset test helps to ensure that the splay tree remains balanced after an operation is performed.

The standard implementation has good amortized complexity, where the overall cost of a sequence of operations is spread out evenly, resulting in better average-case performance. Whereas, the naive implementation may not consider amortized analysis, leading to a higher average cost for individual operations.

The graph suggests a logarithmic behavior for the standard implementation, as the lines representing different subset sizes tend to remain relatively flat and close to each other. This indicates that the time complexity scales logarithmically with the set size (n).

The naive implementation is a simpler and less optimized version of the algorithm, while the standard implementation incorporates additional optimizations and techniques to improve the performance.

The standard implementation takes advantage of the properties of the splay tree, such as the ability to quickly access, frequently accessed nodes and rebalancing the tree after operations. By splaying (rotating) the nodes during the subset test operation, the standard implementation can bring the relevant nodes closer to the root, reducing the overall number of comparisons required in contrast the naive implementation does not leverage these properties, leading to a higher number of operations.

The peak in the graph (of naïve implementation of subset size 1000) represents that the splay tree is a self-adjusting tree and it is designed for efficient access to data elements based on their key values. Now, during the subset test when a sequence of operations is performed such as insertion, deletion and searches; the elements are accessed, they are splayed and brought to the root of the tree. When accessing a subset of elements in splay tree, some elements become more frequently accessed than others and splayed to the root. This creates a temporary imbalance in the tree or corresponds to a temporal imbalance. Basically, peak represents the dynamic self-adjustment of splay trees as frequently accessed elements are moved to the root, temporarily affecting the tree’s balance. And the peaks are only in the higher subsets because of the inefficiencies of the naive algorithm becoming more pronounced as the set size increases. The standard implementation does not exhibit such peaks, indicating better scalability.

The difference in performance across different subset sizes is more noticeable for the naive implementation because the naive algorithm's inefficiencies become more apparent as the set size grows, leading to a larger gap in performance compared to the standard implementation, which is more optimized and can handle larger set sizes more efficiently.

The amortized cost here is generally low, but during subset it can lead to peaks. But, the overall performance of the tree remains efficient.

For all subset sizes, the standard implementation outperforms the naïve implementation, and the difference in performance becomes more pronounced as the set size (n) increases. So, the standard implementation is more efficient than the naïve. Additionally, in both implementations, as the subset size increases, the average number of operations per operation also increases. This behavior is expected because larger subset sizes require more operations to perform the subset test.

**UPDATE:**

The naïve implementation here, is performing better than the standard implementation as seen in the graph.

The graph shows that the average number of operations per operation increases logarithmically with respect to the set size (n). This logarithmic behavior is a characteristic of efficient set implementations that use data structures like hash tables or self-balancing binary search trees, which have logarithmic time complexity for operations like insertion, deletion, and lookup.

For the subset size = 1000, the Standard and Naïve implementations intersect because the performance of the two implementations happens to be very similar.

The intersection is due to constant factors that differ between the two approaches, but the overall logarithmic trend remains consistent across different subset sizes.

{Theorem from Lectures; page 28 & 29}

It is still an open question whether Splay trees are dynamically optimal — that is, at most O(1) times slower than the best possible dynamic tree which knows the access sequence beforehand and adjusts itself optimally.  
  
Theorem (Working set bound): Consider a Splay tree on a set of n items. Access to items x1, . . . , xm are processed in time O(n log n + m + Pi log(1 + zi)), where zi is the size of the working set for xi .