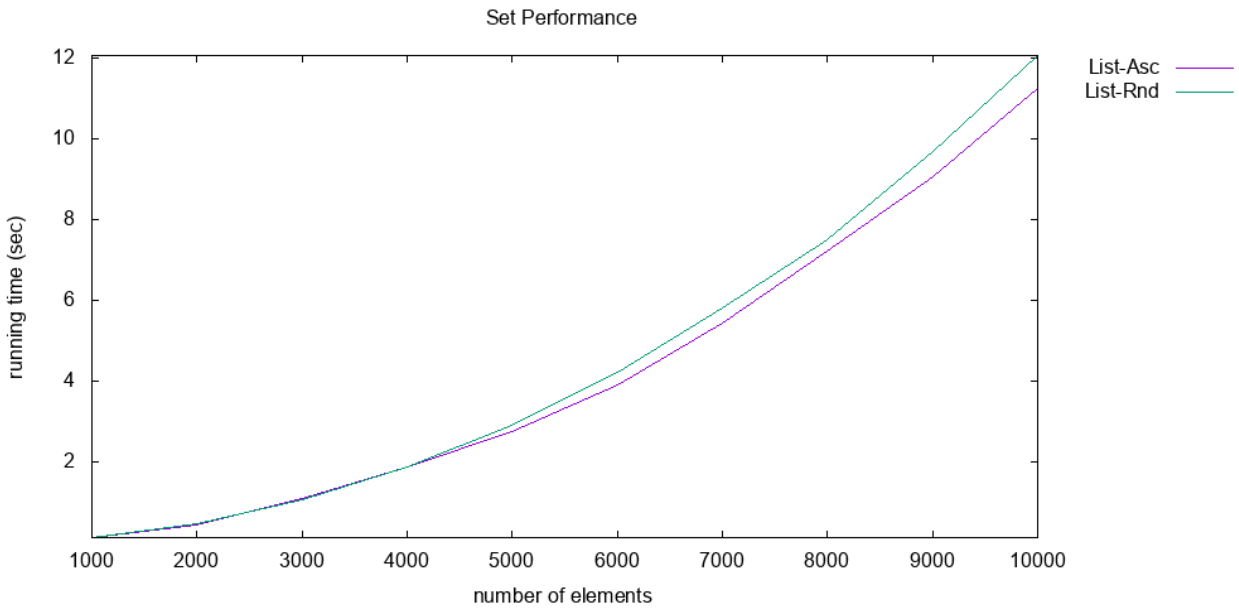
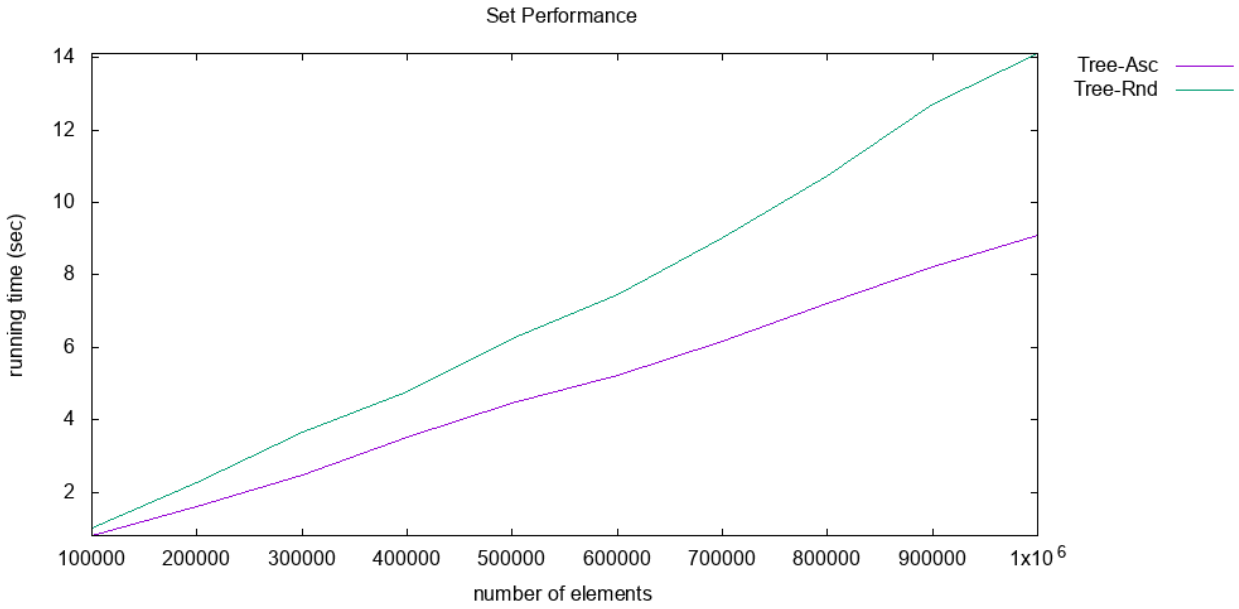




The first graph shows the performance of list sets and the second shows the performance of tree sets. The y-axis is execution time in seconds, and is around 0-12 seconds for list and 0-14 seconds for tree; whereas the x-axis is the number of elements and it ranges from 1000 to 10000 for lists while the number of elements ranges from 100000 to 1000000 for trees.

The execution time being measured is the time it takes to do Insert n randomly chosen integers into the set, do $4n$ membership tests, and delete all n elements in a random order for the list implementation, but **we only have the first two for tree implementation at this stage**. There are two workloads, the n integers were sorted in the ascending order before being inserted in the first one while they remain in the random order in the second one. every data point reported in the graph is actually the median of three repetitions of the experiment.

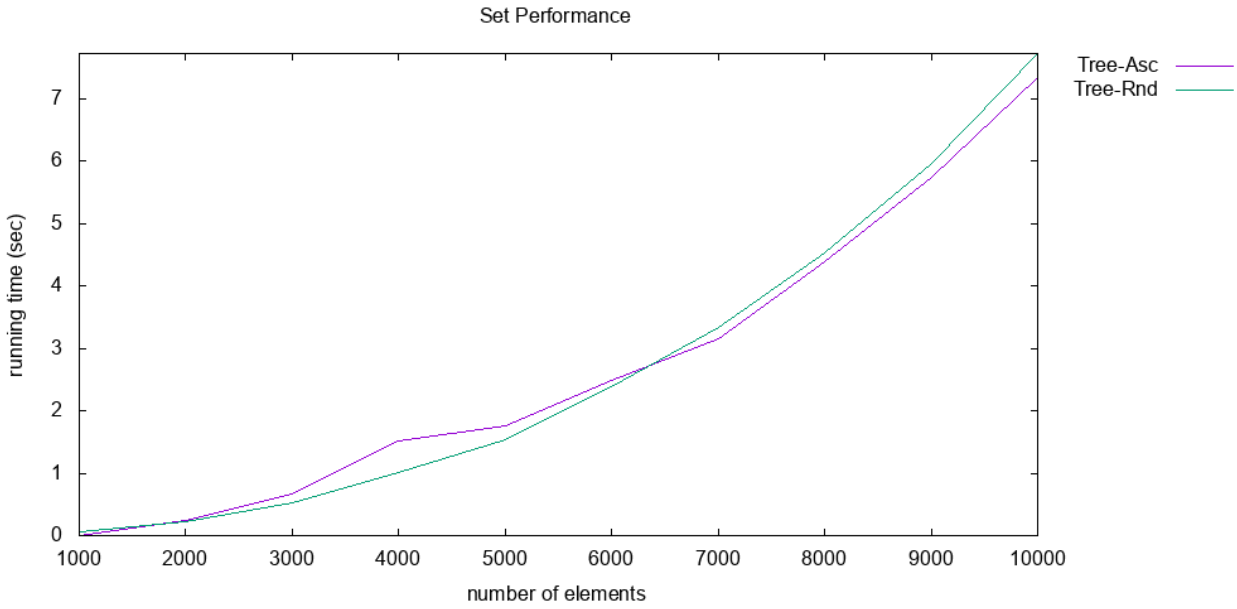
The list graph is consistent with the quadratic complexity by its pattern, and it should have this complexity because insert, $4n$ membership test, and remove all require $O(n)$ operations to be performed and each of the operation is $O(n)$, so there should be $O(n^2)$ work in total. The tree graph is linearithmic, which is much better, and this is because although it still requires $O(n)$ operations for each task, each of the operations will only take O(logn), so there is O(nlogn) work in total. That is also the reason for why that the tree curve is not complete linear, since it is O(nlogn) and should have a slight upward curve .

The performance of list implementation is almost the same for both workloads, and this is probably due to that our implementation of insert does not check the duplicate as we implemented this process in other methods that care about whether there are duplicates in the dictionary, so the order of the elements inserted does not matter as there is no extra operation except for inserting. Also, as the membership testing requires the traversal of the lists in both case and the elements to remove are in a random order, the performance should be similar in both workloads. The graph shows that it performs slightly better in ascending workload, and we believe this is just because of random noise.

The tree implementation does worse for the random workload and that's probably because it triggers more rebalancing operations when doing the insertion.

In both cases, the performance of two workloads are similar overall.

We wrote helper functions in `performanceTest.ml` that perform the workloads and do the timing. We manually recorded the result into the CSV files. We then used `gnuplot` to create graphs from the CSVs.



The above graph shows the performance of the tree implementation after adding the remove task. We cannot make it run on the original as it would become too slow and it seems like that we fail to make the performance linearithmic. We just made the graph based on the performance on the scale of list implementation, which is from $1000 - 10000$. It performs slightly better than the list and looks like $O(n^2)$ and the performance for the two workloads are similar. Most of time are taken for doing remove. We think the potential reasons for why that the implementation of remove cannot hit the expected performance might be that the `remove_bb` helper function is not implemented with its expected efficiency, and a possible improvement could be finding some

more generalized cases for this helper function to implement it more efficient.