COMP90077 – Assignment 1

James Barnes, 820946 – Semester 1, 2020

Experimental Environment

All experiments were performed on a Dell XPS 13 (9360), sporting an Intel Core i7-8550 CPU @ 1.80 GHz with 8GB of onboard memory, running Windows 10 Home 19041.153. The C code, conforming to the 1999 C standard, was compiled using GCC (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0, inside the Windows Subsystem for Linux, running Ubuntu 18.04.

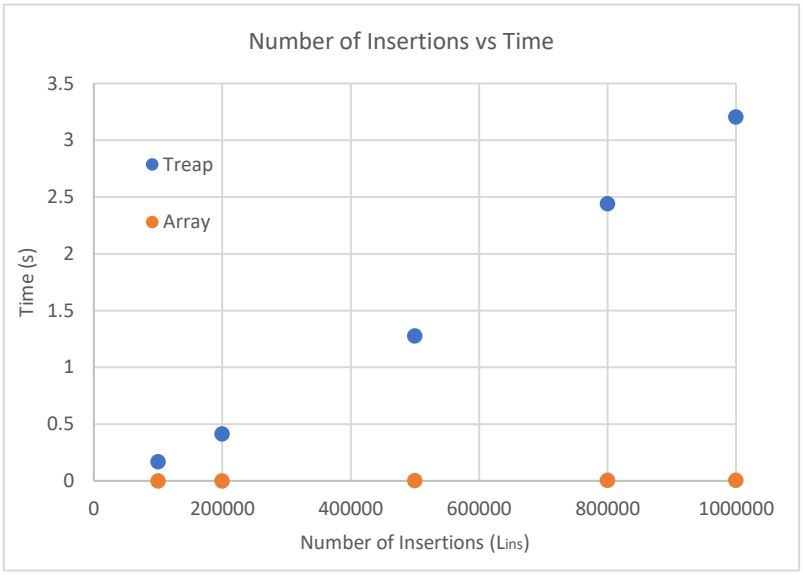
Data Generation

All necessary data for a single given run of any test was created before any timing took place. This was to allow for the algorithm's time alone being the sole contributor to the runtime. This was done by first creating an array of so called `op_t`s (operations), forming the necessary sequence of operations that the test was to follow. The insertion operation generation created a new element to insert into the data structure, and the deletion and search operation generation create an integer key to either delete or search for in it's given operation, following the method described in the provided assignment specification. To support the requirement that the key returned in a deletion operation generation will differ if the chosen element has already been deleted, each element contains a Boolean flag, `deleted`, to check if this element has already been deleted; further, the generator maintains a static array of generated elements used to quickly check if an element has been deleted, indexed by the id of the elements. The random numbers generated are generated such that there is as little skew to the data as possible using the standard C function, `rand`.

Experiments

All tests were performed using the same set of seeds for the random function, meaning that the sequence and values of all operations were the same. These tests are then performed over 50 different seeds, in order to get a better average picture of the runtime of these experiments. Only the average times are shown in graphs, for sake of clarity.

Experiment 1 – Number of Insertions

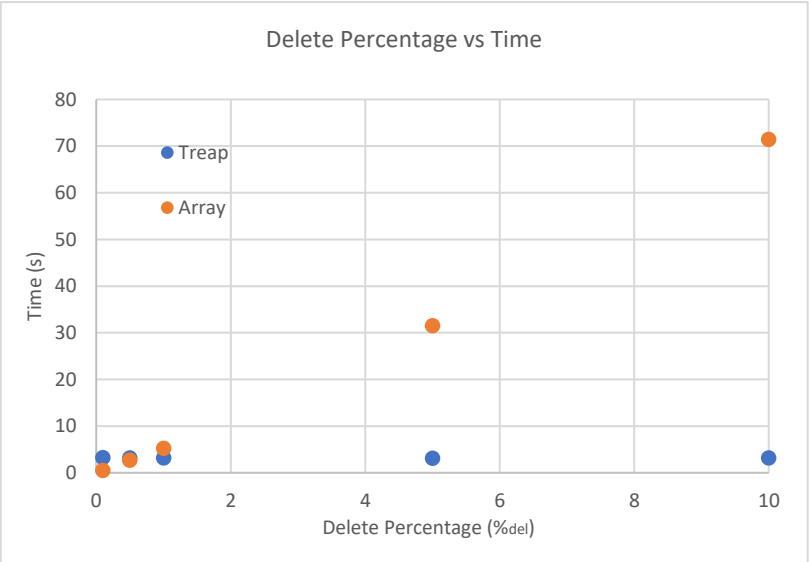


The dynamic array reined superior over the treap in the first experiments.

For insertion, the dynamic array follows $O(1)$ (amortised) time complexity. This is shown in the graph with the array appearing to run almost instantly, however the numbers do show that there is an approximately linear increase in the total runtime, being $O(L_{ins})$.

For the treap, the runtime appears to follow a loglinear curve, which confirms the theory; with L_{ins} insertions, each taking $O(\log L_{ins})$ (amortised) time, the total runtime should follow an $O(L_{ins} \log L_{ins})$ curve as it approximately does.

Experiment 2 – Delete Percentage



In the second experiment, the treap beat out the runtime of the dynamic array.

The treap ran in nearly constant time, as the number of operations was constant, and the runtime of these operations was similar. For the insertion and deletion operations, they both take $O(\log n)$ time, so swapping a delete operation for that of an insertion will not greatly affect the runtime.

For the dynamic array, the runtime follows a linear trend. This is due to the insert and delete operations having different runtimes, of $O(1)$ and $O(n)$ respectively. As you increase the percentage of delete operations, you increase the overall runtime.

In general, the runtime of the program would be as follows (with d denoting $\%_{del}$)

$$(1-d)\ n\ \text{cost(insert)} + d\ n\ \text{cost(delete)}$$

For the treap, this is constant with respect to d

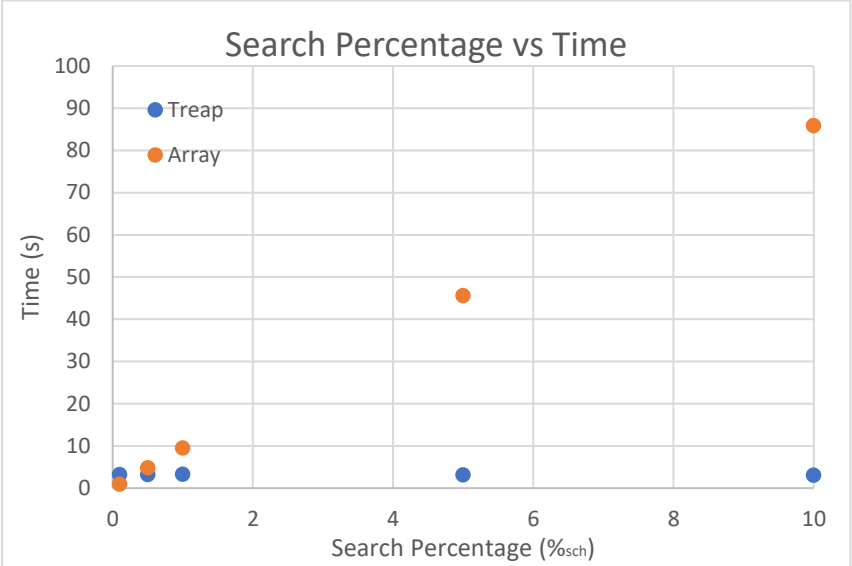
$$(1-d)\ n\ O(\log n) + d\ n\ O(\log n) = O(n \log n)$$

For the dynamic array, this is linear with respect to d

$$(1-d)\ n\ O(1) + d\ n\ O(n) = O(d\ n^2)$$

Experiment 3 – Search Percentage

This experiment shows similar results to experiment 2. As the search and delete operations have the same amortised cost for the treap, and the same amortised cost for the dynamic array, the runtime should behave in a similar way. That being near constant for the treap, and linear for the dynamic array. This is evident with the experimental results.

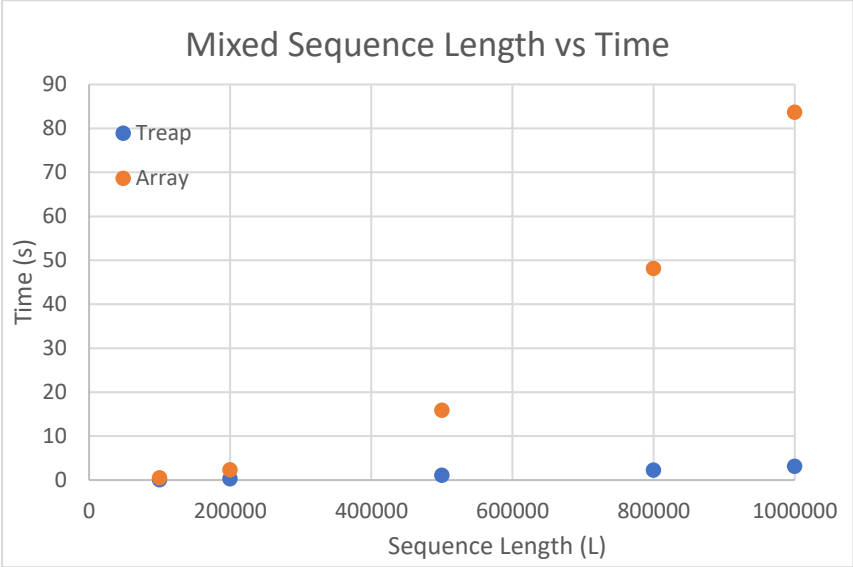


However, with the search operation not reducing the size of the structure, the search operations take more time overall than the delete operations. This is evident with the array especially, where tests can take a substantial amount longer than in experiment 4.

Experiment 4 – Length of Mixed Operation Sequence

Again, the treap reigned superior to the dynamic array in this experiment.

The treap's runtime is dwarfed by the dynamic array's, so the graph does not illustrate this well, however the runtime follows loglinear growth. With all three operations having an $O(\log L)$ (amotised) runtime, the overall runtime should appear to follow $O(L \log L)$.



The dynamic array follows a quadratic runtime in this experiment. This is due to the number of insertions and deletion (both $O(L)$ operations) increasing with the size of the input, L.

In general, as the number of insertions, deletions, and searches is directly proportional to L, the overall runtime of the program is as follows

$$0.9\ L\ \text{cost(insert)} + 0.05\ L\ \text{cost(delete)} + 0.05\ L\ \text{cost(search)}$$

For the treap, this is

$$L\ O(\log L) + L\ O(\log L) + L\ O(\log L) = O(L \log L)$$

For the dynamic array, this is

$$L\ O(1) + L\ O(L) + L\ O(L) = O(L^2)$$

Conclusion

In the first test, the dynamic array outperformed the treap, however in the three subsequent tests, the treap was the top performer. In real-world scenarios, where sequences are more varied than containing only insertions, as they would include deletions and insertions, the treap should be preferred over the array. This is due to the superior runtime of the treap when more varied sequences of operations are involved, as shown in the second, third, and fourth experiments. However, if the workload consisted entirely of insertions, which I believe is unlikely to occur in real-world situations, the dynamic array should be preferred.