COMP90077 ASSIGNMENT 1 REPORT

Sharan Krishnan, 994103

Introduction

The purpose of this experiment is to measure runtimes of a treap and dynamic array over long sequences of insertion, deletion, and search operations; compare results to theoretical bounds; assess which data structure performs better in which conditions; and suggest use cases in which one data structure is preferred to the other.

Experiment Environment

This experiment was conducted on a Dell Inspiron 7506 2n1 that has 16GB memory, 11th gen Intel i7 CPU clocking 2.80GHz and Windows 11 Pro operating system. It is entirely programmed in C and compiled using GCC version 13.1.0.

Randomness in the experiment was simulated using the $\mathrm{rand}(\,)$ function found in the C standard library ($\mathrm{stdlib.h}$). This generates a random integer within the range $[0, \mathrm{RAND\_MAX}]$ where $\mathrm{RAND\_MAX}$ is a constant whose value is dependent on compiler. For the device and compiler used in this experiment, $\mathrm{RAND\_MAX} = 2^{15} - 1$.

Algorithm runtimes were measured using the $\mathrm{gettimeofday}(\,)$ function available in the $\mathrm{sys/time.h}$ library that enables users to measure current times and time differences to microsecond precision.

Finally, memory allocation and reallocation, including data copying for the dynamic array, was achieved with functions $\mathrm{malloc}(\,)$ and $\mathrm{realloc}(\,)$ in the standard library.

Data Generation Implementation

Since generating random numbers within sufficiently large ranges is non-trivial in C, the data generation module developed offers an auxiliary interface:

$$\mathrm{gen\_random}(\,) = \mathrm{rand}(\,) * (\mathrm{RAND\_MAX} + 1) + \mathrm{rand}(\,)$$

which ensures any integer within the range $[0, (\mathrm{RAND\_MAX} + 1)^2 - 1]$ has a non-zero probability of being generated. For our purposes, this equates to the range $[0, 2^{30} - 1]$ which we deem sufficiently large. Therefore, to produce a random integer within the range $[0, b]$ for $b < 2^{30} - 1$ at any point in the experiment, we call $\mathrm{gen\_random}(\,) \bmod b$, assuming, but not guaranteeing, that the distribution drawn from is uniform.

The $\mathrm{gen\_element}(\,)$ function in the module uses this random process to generate an integer search key in the range $[0, 10^7]$ and store it in an $\mathrm{element\_t}$ type structure of the form $(\mathrm{id}, \mathrm{key})$. $\mathrm{id} = 1$ for the first element produced and increments by one for each consecutive element created, guaranteeing uniqueness. After creating element $\mathrm{x} = (\mathrm{id_x}, \mathrm{key_x})$, the function stores the data in a global, fixed length array A, such that $\mathrm{A}[\mathrm{id_x} - 1] = \mathrm{key_x}$, and terminates by returning x.

$\mathrm{gen\_insertion}(\,)$ calls $\mathrm{gen\_element}(\,)$ to produce a data element x and stores it in an $\mathrm{operation\_t}$ type structure that takes the general form $(\mathrm{OP}, \mathrm{x.key}, \mathrm{x})$, where $\mathrm{OP} = 1$ indicates an insertion type operation is generated. This structure is returned.

$\text{gen\_deletion}(\,)$ randomly draws a key from the global array A and returns an operation_t object $(2, \text{key}, \text{NULL})$. Note that the function does not update A to account for the deletion so calls to $\text{gen\_deletion}(\,)$ may produce keys that have already been deleted.

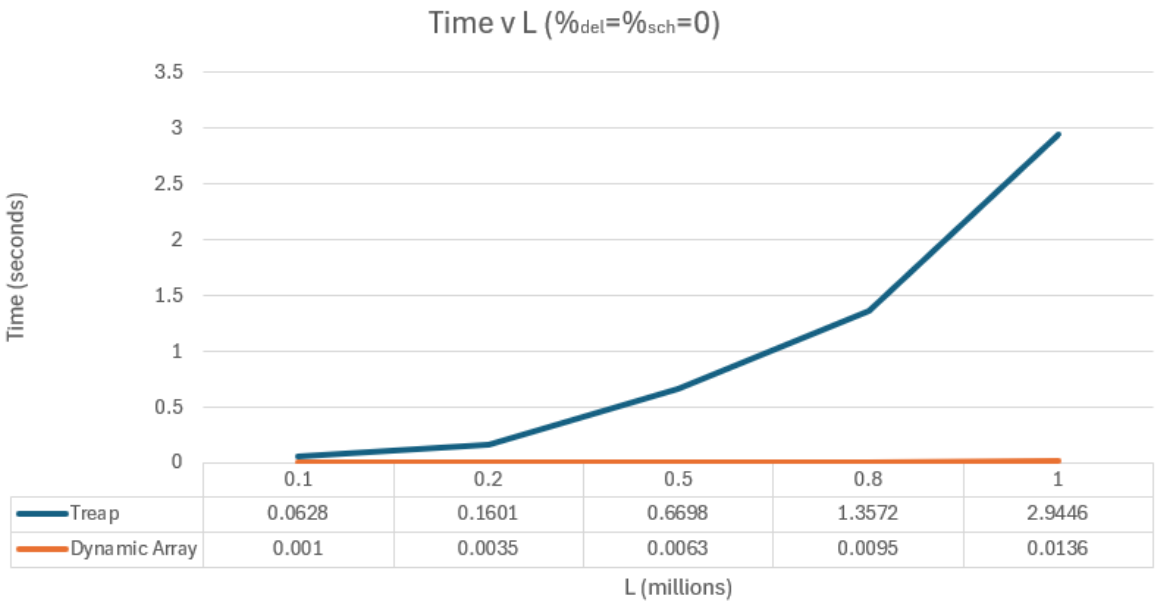
$\text{gen\_search}(\,)$ randomly draws an integer search key from the range $[0, 10^7]$ and returns an operation_t object $(3, \text{key}, \text{NULL})$.

In a separate module, we generate operation_t arrays, σ, of length L where each $\sigma[i]$ was drawn from the following distribution: $p(\sigma[i] \leftarrow \text{gen\_deletion}(\,)) = \%_{\text{del}}$, $p(\sigma[i] \leftarrow \text{gen\_search}(\,)) = \%_{\text{sch}}$ and $p(\sigma[i] \leftarrow \text{gen\_insertion}(\,)) = 1 - \%_{\text{del}} - \%_{\text{sch}}$. L, $\%_{\text{del}}$ and $\%_{\text{sch}}$ are varied over experiments and probabilities are simulated using $\text{gen\_random}(\,)$ calls.

We then iterate through σ and call the respective operations on our treap, measuring the total runtime of processing the entire sequence. For example, if $\sigma[i].\text{OP} == 1$ we call $\text{insert}(\text{treap}, \sigma[i].\text{x})$ and if $\sigma[i].\text{OP} == 2$ we call $\text{delete}(\text{treap}, \sigma[i].\text{key})$. We repeat for the dynamic array.

Results and Analysis

Experiment 1



The figure above shows the total runtime of inserting L elements into both a treap and dynamic array as L increases.

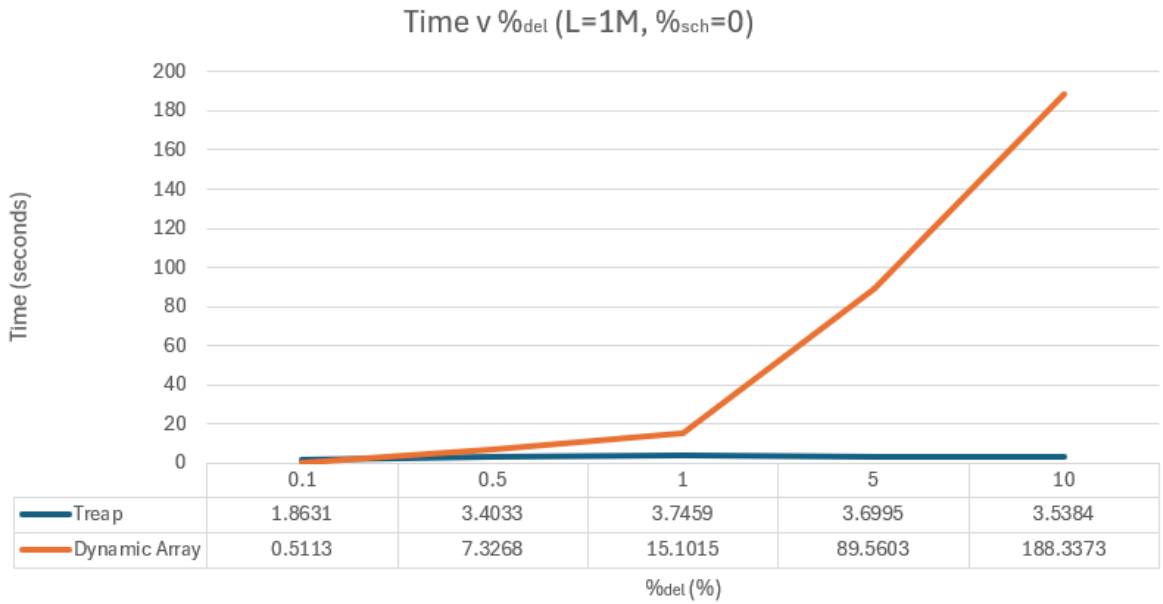
For a treap with n many nodes, the expected cost of insertion is $E(cost(\text{insertion})|n) \leq c \log n \in O(\log n)$ for some $c > 0$, so the total expected cost of L many insertions is bounded by:

$$\sum_{n=0}^{L-1} c \log n \leq c\, L \log L \in O(L \log L)$$

Which explains our observation of super-linearity. The amortised cost of insertion in a dynamic array with n many elements is constant and therefore, over the long sequences we tested, produces an overall $O(L)$ runtime. This is verified by closer inspection of the exact runtime

values of the dynamic array: $\text{array\_runtime}(1\text{M}) \approx 2 \times \text{array\_runtime}(0.5\text{M}) \approx 5 \times \text{array\_runtime}(0.1\text{M})$.

Experiment 2



	0.1	0.5	1	5	10
Treap	1.8631	3.4033	3.7459	3.6995	3.5384
Dynamic Array	0.5113	7.3268	15.1015	89.5603	188.3373

In this experiment, the number of operations stays fixed at 1M and we vary the *expected* amount of deletion operations performed. To simplify the analysis, we will assume that every deletion operation is performed on a key that exists (i.e. the key has not already been deleted) and will ignore the cost of insertion in the array since this is clearly dominated.

Deleting an element from the dynamic array costs linear time with respect to the length of the array because we must first perform a linear search for the corresponding key and may then need to copy elements to shrink the array. After m many operations, the expected number of elements in the dynamic array is $(1 - 2 \cdot \%_{\text{del}})m$. Therefore, the expected cost of a single deletion operation (including shrinkage) is bounded by $c(1 - 2 \cdot \%_{\text{del}})m$ for $c > 0$. The total cost is given by:

$$\%_{\text{del}} \sum_{n=0}^{L-1} c(1 - 2 \cdot \%_{\text{del}})m$$

$$= \frac{c(\%_{\text{del}} - 2 \cdot \%_{\text{del}}^2)L(L-1)}{2} \in O(\%_{\text{del}} L^2)$$

For small enough $\%_{\text{del}}$, $\%_{\text{del}}$ dominates $\%_{\text{del}}^2$ and the cost will increase approximately linearly with $\%_{\text{del}}$. We observe this same effect in the data where $\text{array\_runtime}(10\%) \approx 2 \times \text{array\_runtime}(5\%) \approx 5 \times \text{array\_runtime}(1\%)$.

After m many operations the expected number of nodes in the treap is $n = (1 - 2 \cdot \%_{\text{del}})m$. The expected cost of insertion and deletion is bounded by $c \log n$ for $c > 0$. Therefore, the total expected cost of L many operations is:

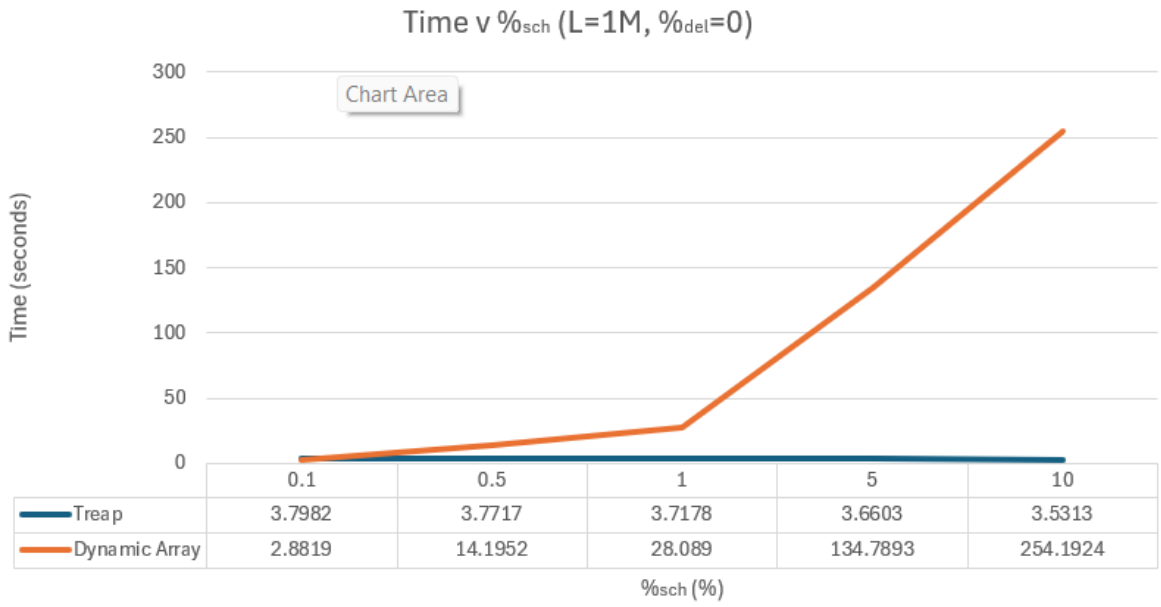
$$\%_{\text{del}} \sum_{m=0}^{L-1} c \log\left((1 - 2 \cdot \%_{\text{del}})m\right) + (1 - \%_{\text{del}}) \sum_{m=0}^{L-1} c \log\left((1 - 2 \cdot \%_{\text{del}})m\right)$$

$$= \sum_{m=0}^{L-1} c \log\ (1 - 2 \cdot \%_{\text{del}})m$$

$$\leq cL \log(1 - 2 \cdot \%_{\text{del}})L \in O\big(L \log\big((1 - 2 \cdot \%_{\text{del}})L\big)\big)$$

According to our derivation, the theoretical upper bound on the total expected cost decreases with $\%_{\text{del}}$, assuming only successful deletions, because the number of nodes in the treap decreases as $\%_{\text{del}}$ increases. Surprisingly, this is not reflected in the experiment results, where the treap's runtime over $\%_{\text{del}}$ is approximately constant around 3.5s (except for $\%_{\text{del}} = 0.1\%$ which appears to be an outlier). The possible reasons are that our assumption of only successful deletions is inaccurate; there exists an unconsidered overhead cost to deletion (such as freeing memory) not considered; or variance in the data.

Experiment 3



	0.1	0.5	1	5	10
Treap	3.7982	3.7717	3.7178	3.6603	3.5313
Dynamic Array	2.8819	14.1952	28.089	134.7893	254.1924

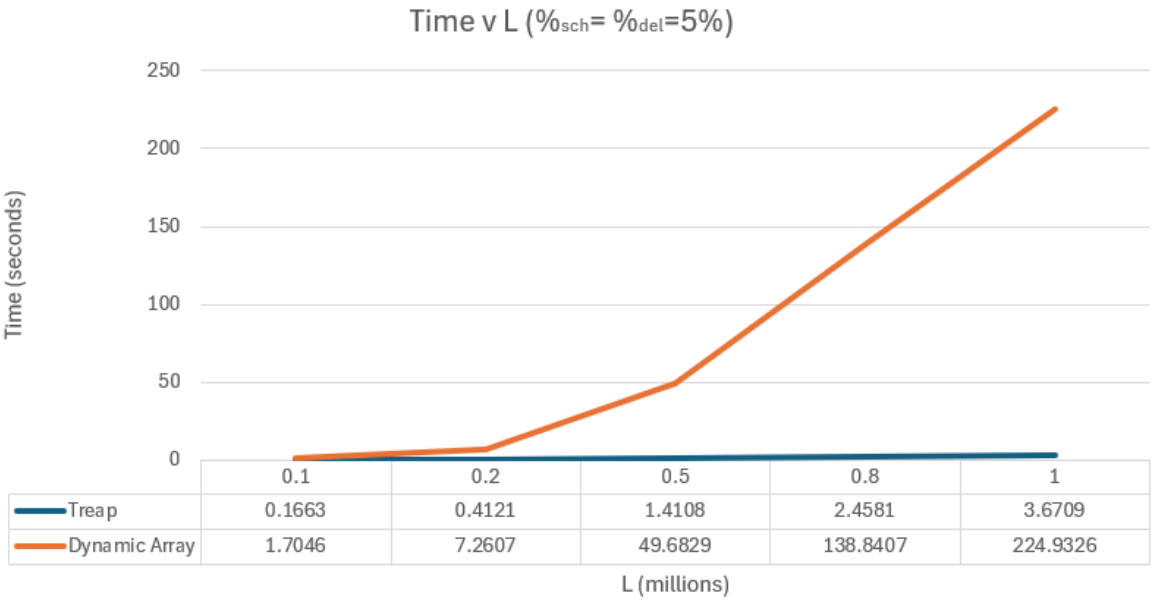
Here we vary the *expected* amount of search operations, $\%_{\text{sch}}$, keeping L fixed. Once again, we will ignore the cost of insertion in the dynamic array since it has a negligible effect on runtime.

After m many operations the expected number of elements in the dynamic array is $(1 - \%_{\text{sch}})m$. By applying the same logic employed in the analysis of experiment 3, if we assume that each search leads to a successful find, then the total cost of search is $c\big(\%_{\text{sch}} - \%_{\text{sch}}^2\big)L(L-1)/2 \in O(\%_{\text{sch}}L^2)$ for a positive c. This is validated by the data since the runtime of the dynamic array increases approximately linearly with $\%_{\text{sch}}$. However, note that search runtime for the dynamic array is notably higher than the deletion runtime observed in experiment 3. This is because the keys searched for are drawn from the range $[0,10^7]$, which is far wider than the range of keys drawn for deletion, so it is much less likely that a search will be successful. We can interpret this as the constant factor c being greater for search than for deletion in dynamic arrays.

The total cost for the treap is $cL \log\big((1 - \%_{\text{sch}})L\big) \in O\big(L \log\big((1 - \%_{\text{sch}})L\big)\big)$ which decreases as $\%_{\text{sch}}$ increases due to the smaller number of nodes. Like experiment 3, we do not see a

noticeable reduction in runtime as $\%_{\text{sch}}$ increases and instead, runtime remains constant at around 3.5s. The reason why is unclear.

Experiment 4



	0.1	0.2	0.5	0.8	1
Treap	0.1663	0.4121	1.4108	2.4581	3.6709
Dynamic Array	1.7046	7.2607	49.6829	138.8407	224.9326

In this final experiment, we measure treap and dynamic array runtimes with respect to varying operation sequence lengths L.

In experiment 2 and 3, we found that total cost functions were in $O(\%_{\text{del}}L^2)$ and $O(\%_{\text{sch}}L^2)$, therefore, the cost with respect to L and constant $\%_{\text{sch}}$ and $\%_{\text{del}}$ is $O(L^2)$. This is verified by the data: $\text{array\_runtime}(1\text{M}) \approx 2^2 \times \text{array\_runtime}(0.5\text{M}) \approx 5^2 \times \text{array\_runtime}(0.1\text{M})$. Additionally, the total cost function is approximately linear in $\%_{\text{del}}$ and $\%_{\text{sch}}$ and in this experiment, half of all non-insertion operations were search and the other half deletions (in expectation). At 1M operations, the runtime of the dynamic array is approximately the midpoint of its runtime in experiment 2 and 3 when 10% of operations are non-insertions, which is exactly what our theoretical analysis suggests.

Similarly, we find the that the total cost for running a sequence of L operations on the treap for fixed $\%_{\text{del}}$ and $\%_{\text{sch}}$ is $O(L \log L)$ which matches the theoretical bound of experiment 1 and the growth of runtimes measured in the two experiments are comparable.

Conclusion

The results empirically validate the theoretical bounds derived for the treap and dynamic array. In particular, the data shows that the dynamic array outperforms the treap in insertion but quickly slows down when a non-trivial amount of search and deletion operations are applied. This corresponds to the constant time amortised cost of insertion but linear time cost of deletion and search in the dynamic array. In contrast, the treap's performance remains relatively constant as insertion, search and deletion frequencies vary, corresponding to the fact that these operations all run in $O(\log n)$ in expectation. Therefore, the dynamic array is suitable for use cases where nearly all operations will be insertions, while the treap is the out-performer in all other use cases involving insertion, deletion, and search.