The University of Melbourne
School of Computing and Information Systems
COMP90077 Advanced Algorithms and Data Structures

Assignment 1 - An Experimental Study on Treaps

Marco Marasco 834482

Introduction

The following report explores the nature of Insertion, Deletion, and Search operations for the Treap and Dynamic Array data structures. The experimentation was conducted by running several experiments with varied size of operation sequences and varied probabilities of distinct operations occurring in the operation. The runtimes of each of these experiments with the different variations were recorded and plotted for analysis.

This report will outline the experimentation environment, explain how the data and operation sequences for the experiments were generated, discuss the results, and conclude with a summary of the findings.

Experimentation Environment

For full transparency of the experimentation and in order to assist for reproduction of the results, an outline of the experimentation environment and tools are presented below:

- **Machine:** Dell Optiplex 990
- **Operating System:** Ubuntu 19.10
- **CPU Model:** Intel(R) Core(TM) i7-2600 CPU
- **CPU Frequency:** 3.40GHz
- **Machine RAM Capacity:** 16GB
- **Compiler:** GNU C++ Compiler v9.2.1
- **Programming Language:** C++

Data Generation

To ensure the quality of the experimentation, the data key values are generated pseudo-randomly during runtime. At the beginning of the program that conducts the experiments, the program creates a pseudo-random number generator using the current epoch time value as an initialisation seed. As machines inherently can't generate truly random numbers, this seed allows for a pseudo-random sequence to be generated. By selecting the current time as the seed, it is highly probable that a distinct sequence of keys is generated for each experiment execution.

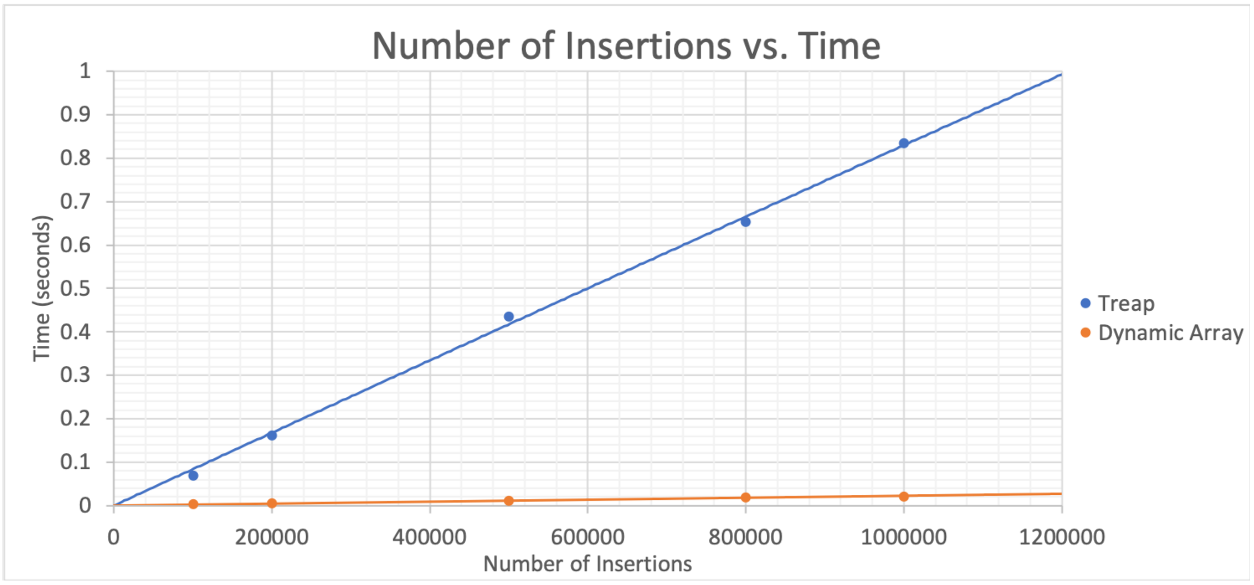
Concerning the sequence of operations for each experiment, as the experimentation requirements only required a rough percentage for the operations in the sequences, each operation was generated by pseudo-randomly selecting a number, and depending if the number fell within a certain pre-defined range, that would dictate what operation type it would be.

The data that is analysed for each experimentation in this report is the average data for 5 separate and distinct executions of the same experiment. This has been done so to mitigate the impact experimental outliers may have on the analysis thus strengthening the accuracy of analysis on the different algorithms for the data structures.

Results

Experiment 1

Figure 1: Experiment 1 Combined Data Structure Results



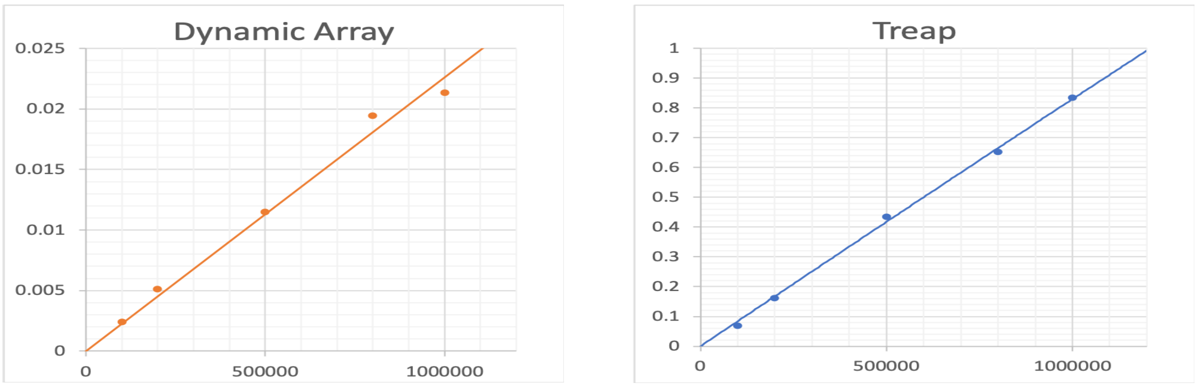


Figure 2: Individual Data Structure Results

Experiment 1 required a sequence of L_{in} insertion operations into the two data structures. From figure 1 it can be seen that for L_{in} insertions, the respective runtimes of the Treap and Dynamic Array diverge rapidly, with the runtime of the Dynamic Array being faster in all cases. The rate of this divergence also indicates a different relationship between the number of insertions and the runtime for each of the data structures. Insertion for the Dynamic Array structures possesses two distinct processes with different associated time complexities. Consider a Dynamic Array A of size N, with n elements currently in the array.

Scenario 1: $n + 1 < N$

In this scenario the Dynamic Array has free space for an insertion, by knowing n, the array allows for an insertion in constant time $O(1)$.

Scenario 2: $n + 1 = N$

This scenario occurs when the insertion causes the Dynamic Array to be full. As by the design of the Dynamic Array, an array is created of size $2N$, and the $n + 1$ elements from the old array are copied across. The cost of the scenario is bound by $O(n)$.

Hence for any given insertion the worst-case complexity is $O(n)$, and for a sequence of n operations, the worst-case complexity would be bound by $O(n^2)$. Looking at figure 2, however, it is evident that the trend line fits a linear relationship.

Amortised time is an effective way to theoretically model the complexity of an algorithm when its worst-case occurs rarely. The amortised time of insertion for a sequence of operations into a Dynamic Array is bound by $O(1)$, hence for n operations an amortised runtime bound of $O(n)$. This bound appears to closer fit the behaviour of the experimentation, with the data have a correlation coefficient of 0.994, indicating a strong, linear relationship (see figure 2).

However, in practice it is not possible to amortise time, but for a simple Dynamic Array it can support experimental analysis. Knowing Scenario 2 only occurs when the number of elements in the Dynamic Array reaches a power of 2 may assist in explaining the anomalous small increase in runtime of 0.001896 seconds between $L_{in} = 0.8 \cdot 10^6$ and $L_{in} = 1 \cdot 10^6$.

Given a size x in the L_{in} values and its next proceeding larger value y:
$|floor(\log_2(x)) - floor(\log_2(y))| \geq 1$ for all sizes x and y, except for $x = 0.8 \cdot 10^6$ and $y = 1 \cdot 10^6$. This is the only size interval whose interval does not cross a power of 2, resulting in the Dynamic Array not undergoing a costly size restructure between these values. The absence of a restructure and resulting minimal shift in runtime assists in further indicating the significant impact restructuring the array has on runtime.

For Treap operations, the expected bound for any operation is $O(log(n))$, hence the runtime trendline would likely follow an $O(n \cdot log(n))$ bound for n insertions. The resulting nature of the Treap trendline in figure 2 support this behaviour.

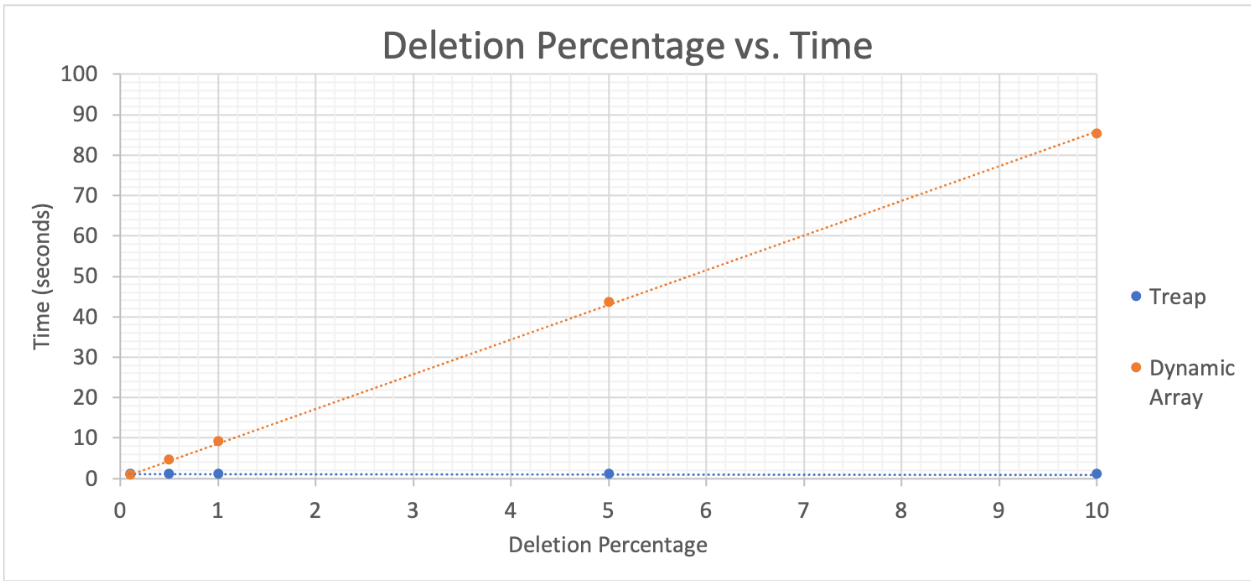
Comparing the two data structures, the Dynamic Array has a far superior algorithm for insertion operations, that has less of a runtime impact as n increases.

From a computational perspective, the process of accessing the next free space in the Dynamic Array is supported by indexing, and allowing insertion in a sequence of operations to be $O(1)$ time, thus in the execution environment, the cost of this action would be extremely small. Whereas in the Treap, each insertion requires multiple reading and writing of elements in the Treap. As the number of insertions grows, the number of these far more costly operations increases. This can explain why the Treap takes a longer time to complete the same number of insertions, but as aforementioned, the Treap's runtime grows at a higher order rate as n increases.

Accordingly, for n operations the Dynamic Array has a lower theoretical bound, and in the experimentation environment its required actions are computationally cheaper, the Dynamic Array outperforms the Treap in terms of runtime.

Experiment 2

Figure 3: Experiment 2 Combined Data Structure Results



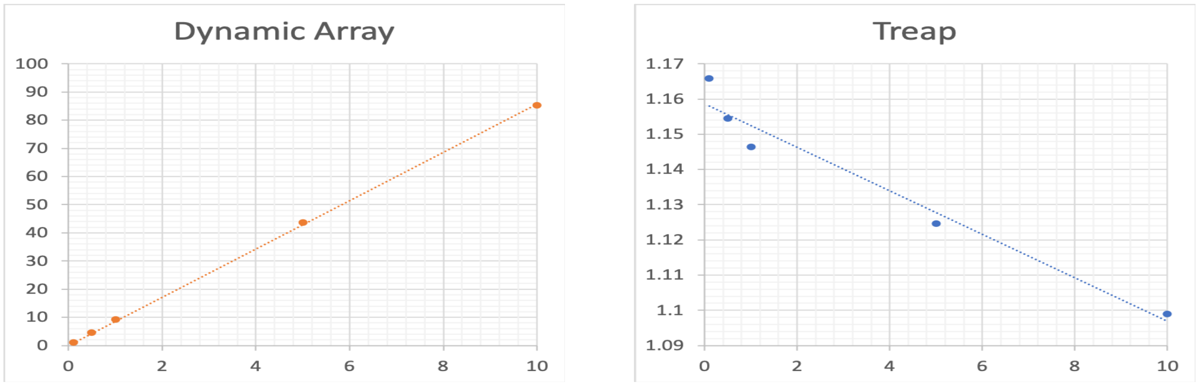


Figure 4: Individual Data Structure Results

Experiment 2 required a sequence of L_{in} insertion and deletion operations into the two data structures. From an initial interpretation of figure 3, the Treap has a significantly better runtime across all the percentages tested.

In Experiment 1, the runtime for insertion on a Dynamic Array was found to be much faster than the Treap. As the superior Dynamic Array runtime has not translated into Experiment 2, it is prudent to analyse the deletion operation of the Dynamic Array to find an insight as to why the runtime is significantly slower.

The Dynamic Array deletion operation operates by performing a linear search for an element, then performing a cheap constant time removal of the element. As in the insertion operation, the deletion operation also resizes the array if the number of the elements becomes less than one quarter the size of the array. This resizing could cause noticeable difference on the runtime, as it did in Experiment 1.

Consider a Dynamic Array A, with size N and it contains n elements, where $n = \frac{N}{2}$. This scenario occurs just after the nth element was inserted, and the size of the array doubled. It is at this moment the experimentation requires the lowest number of deletion operations with respect to N to initiate a resize. To cause a shrink of the array, there needs to be $1 + \frac{N}{4}$ consecutive deletion operations at this moment.

As the percentage of deletions in each of the tested sequences is quite low, as the Dynamic Array grows it becomes increasingly unlikely for enough deletion operations to occur to cause the shrink. It can be deduced searching for the element is what causes the runtime increase. As the sequence order of operations are pseudo-randomly selected, taking each cost operation to be equivalent to the average cost of $O(\frac{n}{2})$ can help shed light on the results.

If the costly search portion of each deletion operation is assumed to take the same amount of time, as the frequency of search operations grows in the sequence, it would be expected that the total runtime of the experiment would grow proportionally to this increase. If completing a sequence of just actual insertion and deletion (no search component) operations has an amortised cost of $O(1)$ (see Appendix 2), for a constant number of operations, the runtimes would theoretically not expect to deviate regardless of the percentage of insertions/deletions in a sequence of n operations.

Looking at the plotted results, the Dynamic Array data appears to follow a linear relationship, with the data having a correlation coefficient of 0.999. This supports the expectation of the runtime growing proportionally to the number of deletions (and therefore searches) in the sequence.

Comparing the results of Experiment 2, the Treap has a significantly faster runtime for the sequence of operations. As was dsicussed in Experiment 1, the expected cost of each Treap operation is bounded by $O(log(n))$, thus we would not expect the runtime results to deviate too significantly from the runtime result for 1 million insertions in Experiment 1.

Closer inspection of the data shows this assumption does not strictly hold, as there is a slight decrease (see Appendix 1, Table 2) in runtime as the percentage of deletion operations increase.

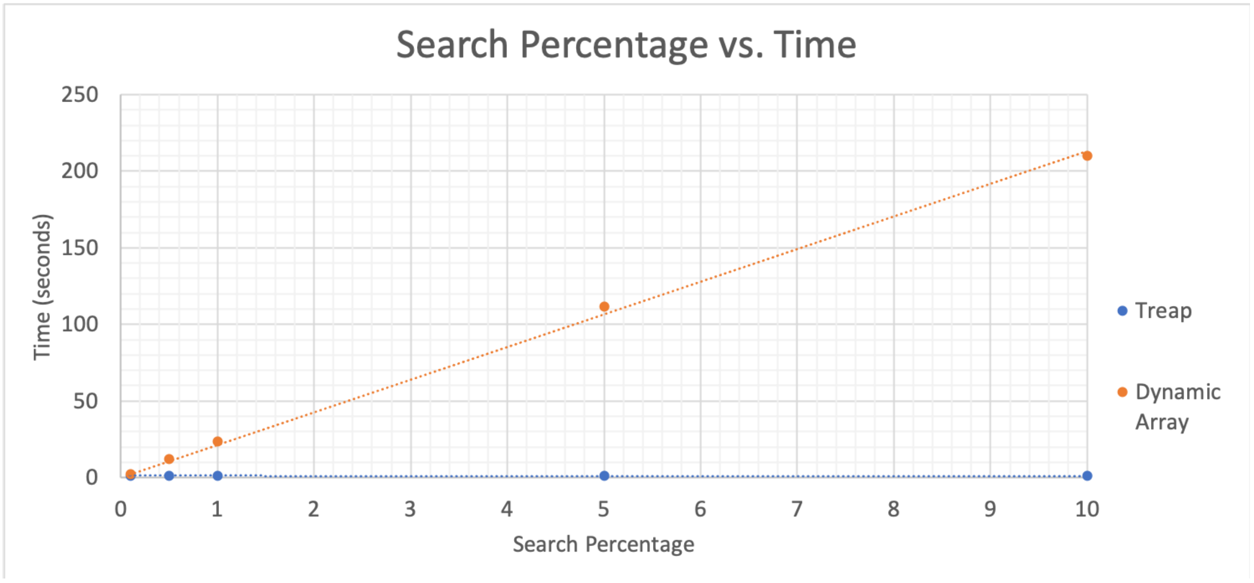
As both insertion and deletion have the same expected runtime, the likely cause in the decrease in time is due to the state of the data structure. This decrease in time may be a result of the tree itself shrinking in size. As the number of nodes in the tree decreases, each insertion and deletion become computationally less expensive as the number of nodes traversed in the operations decreases. Whilst the shift in the number of nodes is quite insignificant for any of the given probabilities, another potential factor to this decrease in runtime could be due to deletion algorithm being more efficient in implementation compared to the insertion.

As for direct comparison between the two data structures, the Treap performs in a significantly faster runtime. Whilst the construction of the Treap was found to be slower in Experiment 1, the scale of the insertion runtime difference is almost negligible when looking at Experiment 2. As the search component of the full deletion operation in a Dynamic Array has a complexity bound of $O(n)$, an order higher than the Treap's $O(log(n))$, it is expected and shown that the runtime for the Dynamic Array will be far slower than the Treap, and the difference between these values grows as n grows.

When the percentage of deletions in the sequence was 0.1%, the Dynamic Array performed faster than the Treap (see Appendix 1). As this sequence of operations was 99.9% insertions, the Dynamic Array appeared to use the runtime buffer for 1 million insertions discovered in Experiment 1, resulting in a faster runtime. The two runtimes, however, deviate immediately after this value.

Experiment 3

Figure 5: Experiment 3 Combined Data Structure Results



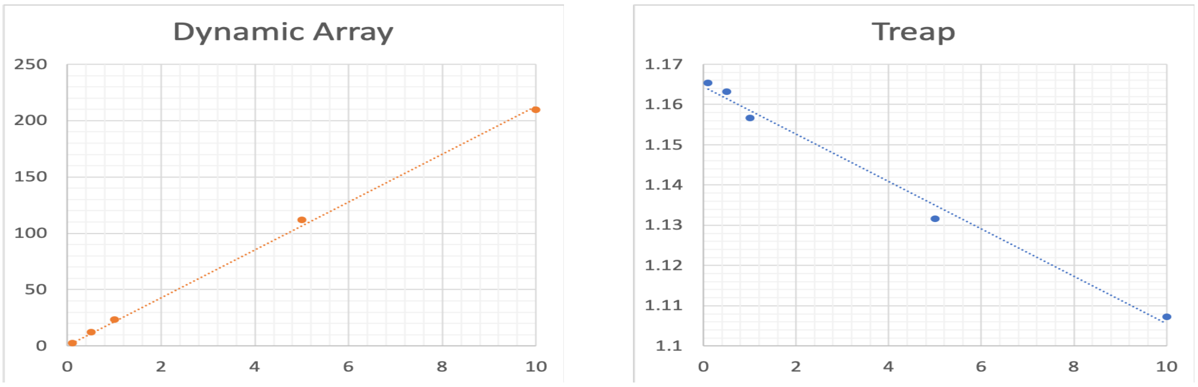


Figure 6: Individual Data Structure Results

Experiment 3 consisted of a sequence of L_{in} insertion and search operations over the two data structures. An initial review of figure 5 showed a significantly longer runtime for the Dynamic Array.

Theoretically, linear search in the Dynamic Array is bound by $O(n)$, hence for a sequence of n operations the worst-case is bound by $O(n^2)$. As the cost of searching for an element in the Dynamic Array is in almost all cases (by Experiment 1) of a higher runtime cost than inserting an element, it is expected that as the frequency of search operations increase in the sequence, so does the runtime.

As was deduced in Experiment 2, nearly all of the cost of deletion was from the linear search, therefore for Experiment 3 which is only concerned with search operation, the nature of the trend for the Dynamic Array in Experiment 3 is expected to be the same as Experiment 2.

First inspection of the data supports this expectation, as the trendline clearly also indicates a positive linear relationship between the percentage of search operations and the runtime. Over the 5 plotted points, the runtime indicated a strong, linear relationship between the two variables, with a correlation coefficient of 0.999.

As all operations of a Treap are bounded by $O(log(n))$, an initial expectation of the results would be that they maintain the same runtime as the same number of operations occurs, as n is constant across all experiment cases. However, the search operation doesn't require any rotations, and as such doesn't require the additional computational costs of reading/writing memory for rotations. This would then indicate that as the number of search operations increased, the runtime would be expected to decrease accordingly.

As the runtime of the search operation in a Dynamic Array $O(n)$ is of a higher complexity order than the Treap's $O(log(n))$, and as the results in Experiment 2 which implicitly completed the search operation during each deletion, it is expected and shown that the runtime for the Dynamic Array will be far slower than the Treap, and the difference between these values grows as n grows. Again, it is worth nothing that when the probability of deletion was 0.1%, the Dynamic Array performed faster than the Treap (see Experiment 2 analysis for reasoning).

The Search operation for both data structures is a subset of the full deletion operation for each data structure respectively. It would then follow for the same deletion/search probabilities, the runtime of the search sequence would be less as it is computationally less expensive. The data contradicts this statement, as in call cases the Experiment 3 execution was slower than the Experiment 2 execution.

As per the specification of the data generator, for the deletion experiments, the generator pseudo-randomly selects the ID of an element for deletion that had definitely been inserted, and in the case it had already been deleted, pseudo-randomly selects an element key. Conversely in generating a value to search, the generator pseudo-randomly selected an element key. It should be noted that the element key range is $[1, 10^7]$, compared to the ID range of $[1, 10^6]$.

From these two methods, it can be seen the deletion generation process has a significantly higher probability of selecting an element that exists in the structures, compared to the search generation process. What this in turn means for the experiment results, is that Experiment 3 is more likely to more search operations for keys that do not exist in the data structure. This in term means more worst-case search operations for the data structures.

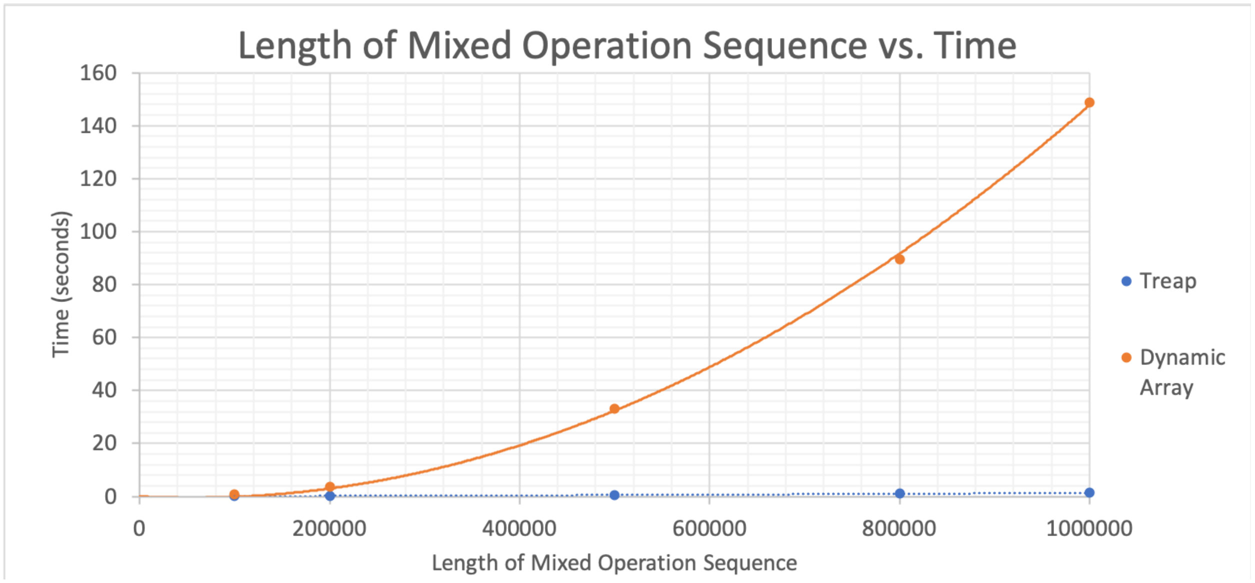
Looking at the Dynamic Array first, the difference between the results in Experiment 3 to Experiment 2 increases as the frequency of the search/deletion operations increases. As Experiment 3 is much more likely to possess worst-case search operations, Experiment 3 is expected to have a higher runtime compared to Experiment 2.

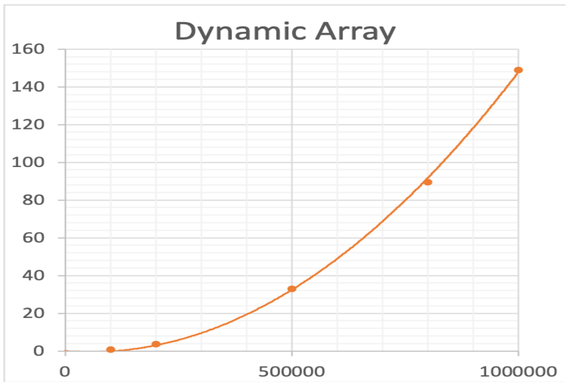
Looking at the Treap structure, whilst the runtimes were slower than Experiment 2, they deviated by no more than 0.5 seconds. As in a randomised Treap, operations are bound by $O(log(n))$, it is unlikely repeated worst-case searches will impact the runtime to the extent of a Dynamic Array. This in turn would suggest little to no difference between Experiments 2 and 3, and furthermore suggest that the Treap will be better able to handle these worst-case search operations compared to the Dynamic Array.

From Experiment 3, it can be concluded the Treap search function is far superior to the Dynamic Array, and particularly the it's ability to handle worst-case searches is superior.

Experiment 4

Figure 7: Experiment 4 Combined Data Structure Results





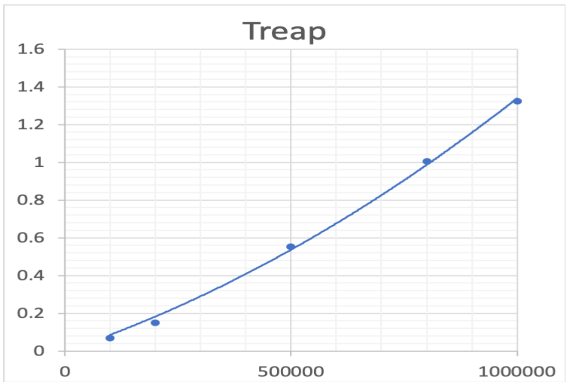


Figure 8: Individual Data Structure Results

Experiment 4 was concerned with the relationship between $L_i n$ operations of search, insertion, and deletion, and the total runtime for performing these sequences on the two data structures.

The results from Experiment 1 confirmed that insertion of the Dynamic Array was faster than the Treap, but Experiments 2 and 3 demonstrated that whatever runtime advantage the Dynamic Array had from insertion was completely removed for search and deletion operations.

For the Dynamic Array, an execution of n operations would be bound by worst-case $O(n^2)$. Experiments 2 and 3 showed most of the runtime for each of their respective executions of n operations was largely made up of the $O(n)$ linear search. Furthermore, the runtime for just performing insertions was insignificant compared to deleting and searching. This would indicate the runtime for Experiment 4 to be dominated by $O(n)$ deleting and searching. Therefore, as n increases, the trendline for runtime would be expected to tend towards $O(n^2)$. In Figure 8, it is clear the trendline closely fits a quadratic trendline. Pleasing, experimental results results matches the expected theoretical behaviour of the Dynamic Array.

Moving analysis to the Treap, as in Experiment 1, the expected runtime trendline for Experiment 4 is expected to follow $O(n \cdot log(n))$. All operations in the Treap have an expected cost of $O(log(n))$, hence graphing n against the runtime would result in trendline following $O(n \cdot log(n))$. The Treap trendline in figure 8 confirms this theoretical behaviour, confirming that regardless of the frequencies of each Treap operation type, the runtime is expected to grow by the bound of $O(n \cdot log(n))$.

Solely considering the results of the 3 earlier experiments, it was expected that Treap would outperform the Dynamic Array as its slower insertion runtime for 1 million elements is negligible compared to the Dynamic Array's deletion and search runtime for any percentage of those operations in a sequence.

Moreover, as the Dynamic Array trendline was expected to tend towards an $O(n^2)$ line, compared to the Treap with an expected trendline behaviour of $O(n \cdot log(n))$, as n increased it was expected the trendlines to deviate further from each other.

Both of these statements are verified by the results in Experiment 4, as figure 7 clearly shows the Treap has a far superior runtime than the Dynamic Array, but also the rate of change of Treap trendline is far less, indicating that the number of operations for a Treap is not expected to increase runtime by the same magnitude as the Dynamic Array.

The findings from Experiment 4 demonstrate that for a sequence of n insertion/deletion/search of varying frequencies, using a Treap to store the data results in far faster overall runtime compared to using a Dynamic Array.

Conclusion

Experiment 1 showed that the construction of a Dynamic Array was faster than the Treap, and the runtime was less impacted as the number of elements grew. The costs in the execution environment for performing the necessary steps in the Dynamic Array's insertion algorithm were far less than the Treap's, resulting in a quicker runtime. Moreover, the trendline for each data structure indicated that increasing the size of Dynamic Array made the runtime grow linearly, whereas the runtime for the Treap followed a trendline of $O(n \cdot log(n)$.

Experiment 2 showed that for a sequence of 1 million operations of insertions and deletions, as the percentage of deletion operations in the sequence grew, the Dynamic Array's runtime increased proportionally, whereas for the Treap, the runtime slightly decreased to due external factors. From this, it was drawn that deletion in a Dynamic Array is far more runtime costly than deletion in a Treap.

The results from Experiment 3 largely demonstrated the impact of searching, but most significantly, worst-case searching in each of the data structures. The results indicated that searching in a Treap was not only faster than the Dynamic Array, but also was better able to handle worst-case search operations.

Experiment 4's results highlighted how each data structure would perform if it were to be used in a practical setting where all three of the tested operations were used. The results of this experiment proved the Treap was a far superior data structure to use. By the nature of its implementation, it is far better designed to handle all three operations, and as the number of operations grew, the runtime was followed a $O(n \cdot log(n))$ trendline, resulting in relatively smaller increases in runtime. Analysing the Dynamic Array, the Dynamic Array was an inferior data structure for Experiment 4, as the trendline that was plotted tended towards $O(n^2)$ behaviour.

Overall, the findings of this report would advise for scenarios where insertion, deletion, and search are required, implementing a Treap would be the better data structure to reduce runtime. For future experimentation, implementing a Dynamic Array that sorts itself after each operation may provide a different perspective on the viability of the data structure, as this would reduce search operation costs, but analysis would need to be made on the tradeoff of regular sorting.

Appendix 1

Number of Insertions	**Treap (s)**	**Dynamic Array (s)**
100000	0.068001	0.002416
200000	0.161811	0.005123
500000	0.434996	0.011501
800000	0.652822	0.019432
1000000	0.833726	0.021328

Table 1: Experiment 1 Results

Deletion Percentage	**Treap (s)**	**Dynamic Array (s)**
0.1	1.165915	1.017438
0.5	1.154539	4.610484
1	1.146332	9.188385
5	1.124586	43.73462
10	1.098937	85.26308

Table 2: Experiment 2 Results

Search Percentage	**Treap (s)**	**Dynamic Array (s)**
0.1	1.165412	2.442142
0.5	1.163193	12.11452
1	1.156643	23.56089
5	1.131646	111.9218
10	1.107294	209.8942

Table 3: Experiment 3 Results

Number of Operations	**Treap (s)**	**Dynamic Array (s)**
100000	0.068203	0.771316
200000	0.149095	3.514123
500000	0.552463	33.03334
800000	1.005919	89.60841
1000000	1.324513	148.9199

Table 4: Experiment 4 Results

Appendix 2

Starting from an empty Dynamic Array, any sequence of n insertion and deletion operations takes $O(n)$ time.

Consider $\alpha_i(X) = \dfrac{Num(X)}{Size(X)}, \quad \dfrac{1}{4} \leq \alpha_i(X) \leq 1$, which is the state of the Dynamic Array after the i^{th} operation.

S_i and N_i are the number of elements and capacity of the array after the i^{th} operation.

Define the potential function $\phi(X) = \begin{cases} 2Num(X)Size(X) & \text{if } \alpha(X) \geq \frac{1}{2} \\ \frac{Size(X)}{2} - Num(X) & \text{if } \alpha(X) < \frac{1}{2} \end{cases}$

Suppose the i^{th} operation is an insertion, the possible cases are:

Case 1. $\alpha_{i-1}(X) \geq \dfrac{1}{2} \Rightarrow \alpha_i(X) \geq \dfrac{1}{2}$

$S_{i-1} = \dfrac{S_{i-1}}{2}, N_{i-1} = N_i - 1$

$$\begin{aligned}
\text{Amortised cost} &= c_i + \phi_i - \phi_{i-1} \\
&= N_i + 2 \cdot N_i - S_i(2 \cdot (N_{i-1})S_{i-1}) \\
&= N_i + 2 - \frac{S_i}{2} \\
&\text{As } N_i = \frac{S_i}{2} + 1 \\
&= 3 \\
&= O(1)
\end{aligned} \tag{1}$$

Case 2. $\alpha_{i-1}(X) < \dfrac{1}{2} \Rightarrow \alpha_i(X) < \dfrac{1}{2}$

$$\begin{aligned}
\text{Amortised cost} &= c_i + \phi_i - \phi_{i-1} \\
&= 1 + (\frac{S_i}{2} - N_i) - (\frac{S_{i-1}}{2} - N_{i-1}) \\
&= 1 + \frac{S_i}{2} - N_i - \frac{S_{i-1}}{2} + N_i - 1 \\
&= 0 \\
&= O(1)
\end{aligned} \tag{2}$$

Case 3. $\alpha_{i-1}(X) < \frac{1}{2} \Rightarrow \alpha_i(X) \geq \frac{1}{2}$

$$\begin{aligned}
\text{Amortised cost} &= c_i + \phi_i - \phi_{i-1} \\
&= 1 + 2 \cdot N_i - S_i - (\frac{S_{i-1}}{2} - N_{i-1}) \\
&= 1 + 2 \cdot N_{i-1} + 2 - S_i - \frac{S_{i-1}}{2} + N_{i-1} \\
&= 3 + 3 \cdot N_{i-1} - \frac{3 \cdot S_{i-1}}{2} \\
&< 3 + \frac{3 \cdot S_{i-1}}{2} - \frac{3 \cdot S_{i-1}}{2} \\
&< 3 \\
&= O(1)
\end{aligned} \tag{3}$$

Suppose the i^{th} operation is a deletion, the possible cases are:

Case 4. $\alpha_{i-1}(X) \geq \frac{1}{2} \Rightarrow \alpha_i(X) \geq \frac{1}{2}$

$$\begin{aligned}
\text{Amortised cost} &= c_i + \phi_i - \phi_{i-1} \\
&= 1 + 2 \cdot N_i - S_i - (2 \cdot N_{i-1} - S_{i-1}) \\
&= 1 + 2 \cdot N_i - S_i - 2 \cdot N_{i-1} - 2 + S_{i-1} \\
&= -1 \\
&= O(1)
\end{aligned} \tag{4}$$

Case 5. $\alpha_{i-1}(X) \geq \frac{1}{2} \Rightarrow \alpha_i(X) < \frac{1}{2}$

$$\begin{aligned}
\text{Amortised cost} &= c_i + \phi_i - \phi_{i-1} \\
&= 1 + (\frac{S_i}{2} - N_i) - (2 \cdot N_{i-1} - S_{i-1}) \\
&= 1 + \frac{S_i}{2} - N_{i-1} + 1 - 2 \cdot N_{i-1} + S_i \\
&= 2 + \frac{3 \cdot S_i}{2} - 3 \cdot N_{i-1} \\
&= 2 + \frac{3 \cdot S_i}{2} - 3 \cdot N_i - 3 \\
&= -1 \text{as} \frac{N_i}{S_i} < \frac{1}{2}, N_i < \frac{S_i}{2} \\
&= O(1)
\end{aligned} \tag{5}$$

Case 6. $\alpha_{i-1}(X) = \frac{1}{4}$, this causes a shrink. $S_i = \frac{S_{i-1}}{2}, \quad N_i = \frac{S_{i-1}}{2}, \quad c_i = N_i + 1.$

$$\begin{aligned}
\text{Amortised cost} &= c_i + \phi_i - \phi_{i-1} \\
&= 1 + N_i + (\frac{S_i}{2} - N_i) - (\frac{S_{i-1}}{2} - N_{i-1}) \\
&= 1 + N_i + \frac{S_i}{2} - N_i - S_i + \frac{S_i}{2} \\
&= 1 \\
&= O(1)
\end{aligned} \tag{6}$$

Case 7. $\alpha_{i-1}(X) \leq \frac{1}{2} \Rightarrow \frac{1}{4} \leq \alpha_i(X) < \frac{1}{2}$

$$\begin{aligned}
\text{Amortised cost} &= c_i + \phi_i - \phi_{i-1} \\
&= 1 + \frac{S_i}{2} - N_i - \frac{S_{i-1}}{2} + N_{i-1} \\
&= 1 + N_{i-1} + \frac{S_i}{2} - N_{i-1} + 1 - \frac{S_i}{2} \\
&= 2 \\
&= O(1)
\end{aligned} \tag{7}$$

Hence for all cases, the amortised cost for insertion or deletion is $O(1)$.