Advanced Algorithms and Data Structures Assignment 1

Student ID: 760625

Name: Tom Harris

March 2020

1 Experimental Environment

These experiments were conducted on a MacBook Pro with a processor speed of 2.2 GHz and a 16GB RAM. The operating system used was macOS Mojave 10.14.6. The programming language used was Java (Java 8 - 1.8.0_202).

2 Data Generation

Each experiment involved testing the runtime of a Treap and a Dynamic array in handling a series of standard operations. Each operation had randomly generated data as inputs that were then submitted to each structure where the appropriate action was taken.

The insertion operation generates a new element (*id* and *key* pair) with a unique integer *id* and a randomly generated integer *key* between 0 and 10^7. The deletion operation draws a random *id* from the generated elements for deletion. If this element has already been deleted, a random *key* is chosen between 0 and 10^7. The search operation randomly draws a *key* between 0 and 10^7 to be searched for.

The experiments had a set of operations generated that varied depending on what aspect of each data structure that was being tested. The total amount of time for each structure to successfully process all the generated operations was then measured.

3 Experiments

Each of the experiments were run 10 times with the average run time of each algorithm in each instance being taken. This was done to reduce the chance of an anomalous result.

3.1 Exp 1: Time vs. Number of Insertions

3.1.1 Result Demonstration

See Figure 1

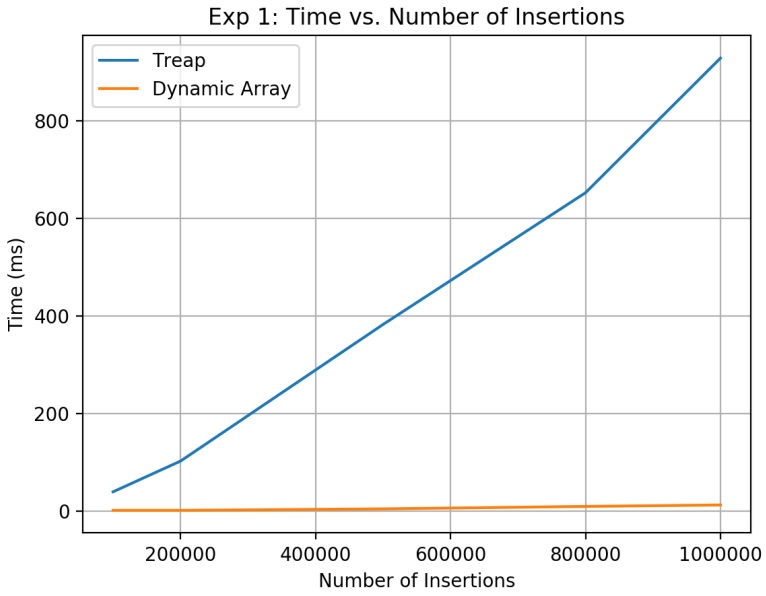

Figure 1: Exp 1 - Time vs. Number of Insertions

3.1.2 Analysis

The first experiment assessed both the Treap and the Dynamic array's performance when dealing with varied length sequences of insertion operations. The Dynamic array performs better in this experiment, significantly outperforming the Treap as number of insertions approaches 10^6. The results show a higher rate of change for the Treap which increases from an initial 40ms required to perform 10^5 insertions to 929ms required to perform 10^6 insertions, while the Dynamic array slowly increases from 2ms to 15ms across the range of insertions required.

This result is expected when the nature of both algorithms and their associated cost when dealing with an insertion operation are considered. Treap insertions have an expected cost of O(logn), where n is the number of elements currently in the data structure. This is obtained from the random assignment of priorities (between 0 and 1) of new nodes when inserting new elements. This expected cost is reflected in the growth of the Treap as number of insertions increases;

the rate of growth increases slightly, but remains mostly consistent. This is expected as the cost of an operation at n=10^5 and n=10^6 only slightly varies, due to the nature of log growth.

The Dynamic array has an amortized cost of O(1) for sequential insertions. This is despite the O(n) worst case cost of each insertion when resizing of the array is required. This amortized cost is due to the large majority of insertions only requiring an assignment of the next cell to the newly generated key. Due to the use of milliseconds in this experiment, it is difficult to accurately determine if the expected linear relationship exists, but a definite steady increase is seen in the range of insertions.

This difference in operation cost between the two algorithm's explains the significant performance difference between the two and the divergence of the two curves.

An important point to mention in this analysis is the difference between amortized and expected cost and what this means for a comparison between them. Expected cost represents the most likely cost of an operation considering the nature of an algorithm being considered. In the case of the Treap, insertions require that the priority of a new node is randomized and the Treap be resolved to satisfy the heap condition; this makes a self-balanced tree a more likely outcome, allowing for this expected cost of O(logn) to be obtained. This does not rule out that a specific run of the experiment could get 'unlucky' and produce a number of insertions that are closer to the worst case run time of O(n) (i.e. a tree that has a height closer to n than logn). However, as expected, with $> 10^5$ insertions the results reflect an O(logn) cost. Amortized cost considers an algorithm's performance over a sequence of operations. In the case of the Dynamic array, a sequence of only insertions are considered. This analysis will provide a more accurate cost bound than the worst case (O(n)) as the sequence of operations means that a worst case scenario can only occur periodically when the array needs to be expanded (resize() function is called).

3.2 Exp 2: Time vs. Deletion Percentage

3.2.1 Result Demonstration

See Figure 2

3.2.2 Analysis

The second experiment assessed both the Treap and the Dynamic array's performance when dealing with varied percentages of deletion operations in fixed length sequence of insertion and deletion operations. The Treap largely outperforms the Dynamic array in this experiment, with the Dynamic array only performing better when the deletion percentage $<$ 1.5%.

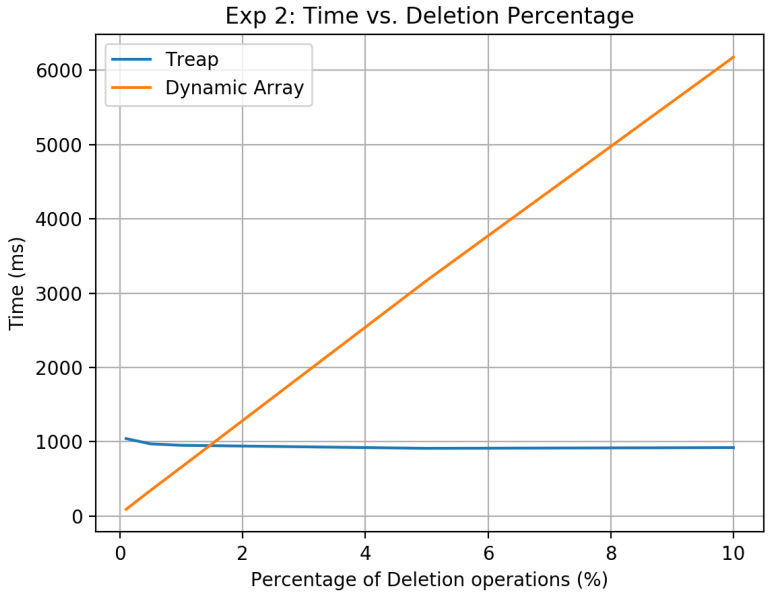

Figure 2: Exp 2 - Time vs. Deletion Percentage

A Treap has an O(logn) expected cost for both insertion and deletion, regardless of the number and type of operations that have preceded the current operation. This means that the variation of the portion of the 10^6 operations that are deletions should have minimal effect on the run time of the total sequence. This is reflected in the consistent total running time of around 1000ms in the Treap.

Dynamic array deletions scan the array for the key to be deleted which gives the operation an O(n) cost. This means that as the portion of the operations that are deletions is increased, the total run time of the Dynamic array algorithm increases. This relationship is approximately linear as the portion of the fixed number of operations that are O(n) cost increase. The initial better performance of the Dynamic array when the portion of insertions $>$ 98.5% can be explained by the results from experiment 1.

The Dynamic array also has a mechanism for 'downsizing' the length of the array if the current contents does not fill 1/4 of the total allocated length of the array. This involves creation of a new array that is half the size of the current array and copying the contents across, similar to the resize() functionality, but for reducing size. This operation is also an O(n) cost, which would further increase the total runtime of a set of operations with a higher proportion of deletions. However, it should be noted that in this experiment, the likelihood of

achieving enough deletions in close proximity to cause such a 'downsize' would be highly unlikely.

3.3 Exp 3: Time vs. Search Percentage

3.3.1 Result Demonstration

See Figure 3

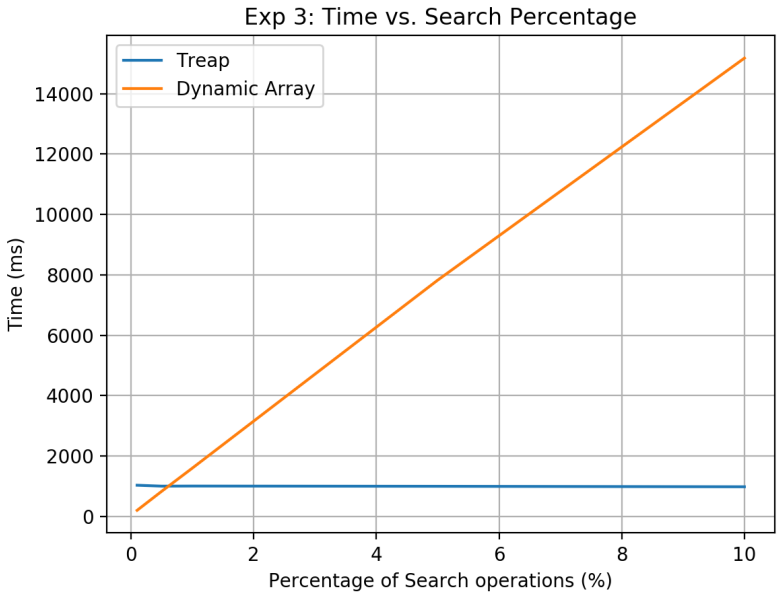

Figure 3: Exp 3 - Time vs. Search Percentage

3.3.2 Analysis

The third experiment assessed both the Treap and the Dynamic array's performance when dealing with varied percentages of search operations in fixed length sequence of insertion and search operations. The Treap largely outperforms the Dynamic array in this experiment, with the Dynamic array only performing better when the search percentage $< 0.5\%$.

A Treap has an O(logn) expected cost for both insertion and search regardless of the number and type of operations that have preceded the current operation. This means, similarly to experiment 2, the Treap consistently achieves a runtime of around 1000ms regardless of the percentage of search operations.

Dynamic array search operations scan the stored keys for a randomly generated key between 0 and 10^7, which is an O(n) operation. This results in a similar relationship to Experiment 2 with a linear increase in runtime as percentage of search operations increases. The magnitude of the runtime will be larger than the deletion experiment (experiment 2) where, although a search is invoked within that, the nature of the generation of the deletion event means that the key is generally from the array already. This means it is more likely that a deletion will not be required to scan the whole array, whereas the search operation often will. This is reflected in the comparison of experiment 2 and 3 outputs; Experiment 2 increases up to 6178ms total runtime for the Dynamic array on the largest percentage, whereas Experiment 3 increases up to 15174ms.

3.4 Exp 4: Time vs. Length of Mixed Operation Sequence

3.4.1 Result Demonstration

See Figure 4

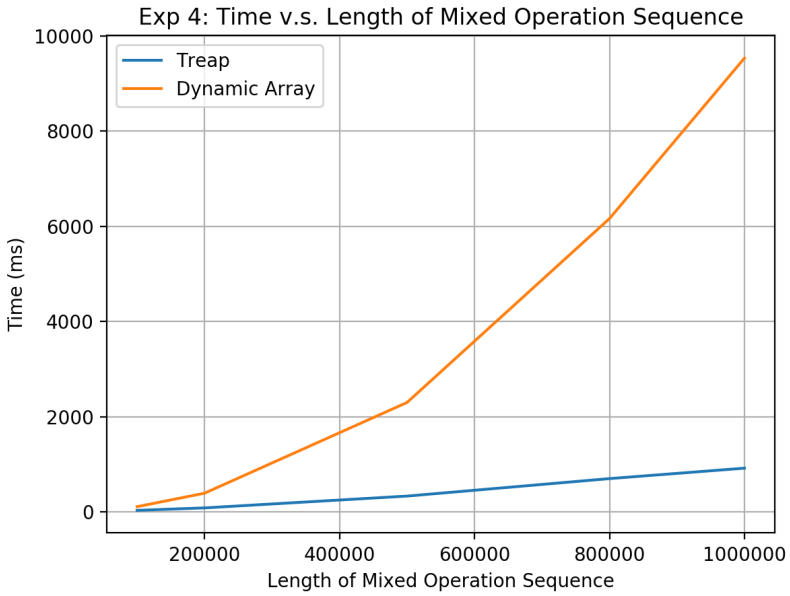

Figure 4: Exp 4 - Time vs. Length of Mixed Operation Sequence

3.4.2 Analysis

The fourth experiment assessed both the Treap and the Dynamic array's performance when dealing with fixed percentages of search and deletion operations in

varied length sequence of insertion, deletion and search operations. The Treap outperforms the Dynamic array in this experiment, with the margin of difference between the two increasing as the length of operations increases.

A Treap has O(logn) expected cost for insertion, deletion and search operations regardless of the number and type of operations that have preceded the current operation. This means when comparing experiment 1 and 4 we should expect similar run times as the total number of operations is varied. This is reflected in the two figures with the 10^6 operations runtime for experiment 1 taking 929ms and experiment 4 taking 922ms.

As shown in experiment 2 and 3, the dynamic array performs significantly worse when deletion and search operations are included. This means that comparing experiment 1 to the experiment 4 results, we should see a significant increase in the total running time of the dynamic array as the number of operations is increased. This is reflected in the two figures.

4 Conclusion

This analysis gave a basic comparison of the Treap and Dynamic array data structures and how they behave in a variety of circumstances. The Treap was shown to be a consistent data structure, being able to continually ensure an O(logn) expected cost for the three basic operations tested (insertion, search, deletion). This meant that as operation variety was altered, the Treap maintained similar total run times (as shown in experiments 2, 3 and in comparing 1 and 4). The Dynamic array's O(1) amortized cost for insertion sequences meant the data structure performed well in insertion only sequences (experiment 1); however, due to the nature of the search and deletion operations, it suffered in the varied operation experiments. This comparison provides a simple demonstration of the strengths and weaknesses of both data structures and provides a small insight into the appropriate circumstances each could be used.