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

Assignment 1, 2020

Karl Flores : 760493

Experimental Analysis on the Runtime of Treap and Dynamic Array Data Structures

Experimental Setup

The Data structures and experiment program were implemented using C, compiled using gcc (c99). Experiments were single threaded and conducted on a 2016 Model Apple Macbook Pro running macOS 10.14 on a 2.7Ghz Intel Core i7 processor.

Data Collection

The experiments conducted on the treap and dynamic array data structures comprised of running three operations on each - inserting an element, searching for an element and deleting an element in the data structure. As such a generator was implemented to generate the required operations. The generator was implemented as follows. Each element had an assoicated id and key. Let X_i be the ith element generated to be inserted, the id's were generated as follows $X_i.id = X_{i-1}.id + 1$ where $(X_0.id = 1)$. A pseudorandom number in the range $[1, 10^7]$ was generated for the search key of X_i. To generate a search key to do a search operation on the data structure, a pseudorandom number was generated in the range $[1, 10^7]$. Finally, to generate the search key of an item to be deleted in the data structure, a pseudorandom number in the range $[1, 10^7]$ was generated – if the corresponding element had not been deleted yet, the search key corresponding to the id of that generated element was returned as the search key for the deletion operation. If the element that was associated with this id was already deleted in the data structure, another psuedorandom number in the range $[1, 10^7]$ was generated as the search key for the deletion operation. *(Note: we assumed that C's rand() function generated a uniform distribution over time)*

Experiments were conducted by generating a sequence of deletion, search and insertion operations to be conducted on both data structures. To seperate the runtime of the generator to the that of the execution of the sequence of operations on each of the data structures, all operations to be executed were generated before the execution and stored in an array. This array of operations/queries to the data structure were then executed on each of the data structures, meaning that the same sequence of operations were executed on each of the data

structures. In order to time the sequence execution, a call to the function gettimeofday() was made before and after inorder to time to the nearest 10^-6 second. Tests were then automated and collected using a python script, which executed the C code for each the experiments with the desired parameters, and the data was collected and plotted using the matplotlib package.

Experiment 1: Time vs Insertion Only Sequences

Description

Insertion only sequences were generated of the following lengths: $0.1M, 0.2M, 0.5M, 0.8M, 1M$ (where M=million), the time to execute each of the sequences was collected and then plotted.

Results

This experiment was run 25 times, on each run, new data was being generated and timed. The means of each runtime were then plotted.

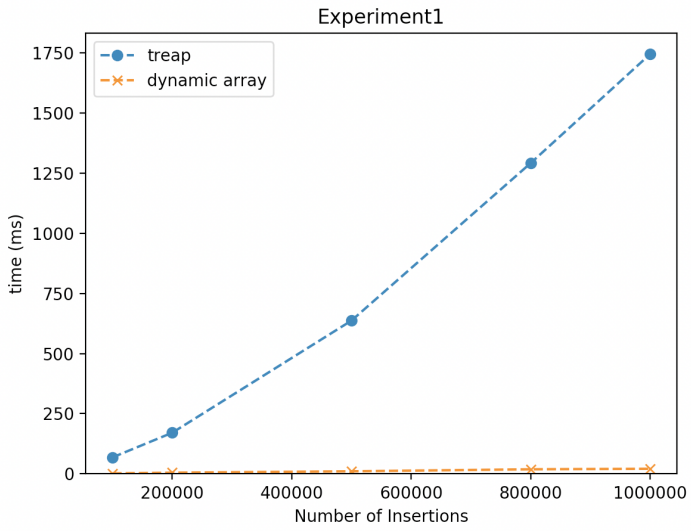

Figure 1: Experiment 1 Results

Analysis

Figure 1 shows that the dynamic array out performs the treap when across all insertion only runs. At the length of 0.1M the total run time on the treap was 3 times slower than that of the dynamic array, and at the length of 1M insertions, the treap was 83 times slower than that of the dynamic array. The difference in the run times can be attributed to the time it takes for each data structure to insert a single element. For a dyanamic array, the amoritized cost of inserting an element is $O(1)$, thus if we consider a sequence of length m insertions, we expect that the total run time to execute would be $O(m)$. A randomized

treap containing d nodes, has an expected height of $\log d$, thus the expected worst case time to insert an element into the treap is $O(\log d)$. Hence considering the same sequence of m insertions, the total run time is expected to be $\sum_{i=1}^{m} \log i$ which we can bound as $O(m \log m)$ (since $\sum_{i=1}^{m} \log i < m \log m$ for $m \in \mathbb{N}^+$). Thus the runtime of treap is expected to be slower than that of dynamic array across all insertion lengths. Figure 1, shows that across all insertion lengths, the dynamic array outperforms the treap substantially, thus supporting our theoretical claims.

Experiment 2: Time vs $\%_{del}$ (Deletion-Insertion Sequences)

Description

Insertion/search sequences were generated with percentage of deletion being varied each run, the time to execute each of the sequences was collected and then plotted.

Results

This experiment was run 5 times, on each run, new data was being generated and timed. The means of each runtime were then plotted.

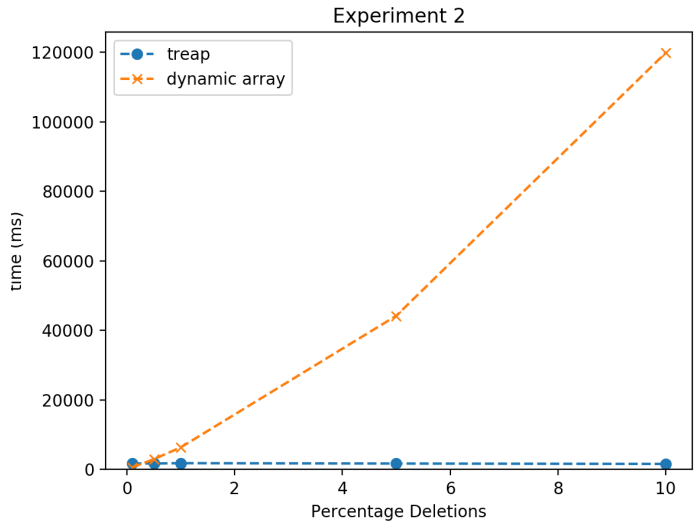

Figure 2: Experiment 2 Results

Analysis

Figure 3 shows that with the exception of the $0.1\%_{del}$ run, the treap out performs the dynamic array when across all insertion only runs. Deleting an element from a treap consists of searching for the element, then rotating that element to the leaf. Thus, if a treap has a height of h, consider an element on at the (i)-th depth. It would take i time to search for this element, then we would need to rotate the element to $(h-1)$-th depth. Thus, as a one

rotation pushes the element 1 depth deeper in the tree, it will take $h - 1 - i$ rotations to push that element to the leaf. Adding the time to find the element and the time taken to rotate the element to the leaf results $O(h)$ time to delete. As the expected height of a treap is $O(\log d)$, deleting an element from the treap takes $O(\log d)$ time. Therefore for a fixed size 1M total insertions and deletions, varying the $\%_{del}$ is not expected to change the overall runtime of the sequence of operations. Figure 2 shows that the experimental runtime of the treap is roughly constant across all $\%_{del}$ runs. For the dynamic array the deleting an element is dominated by the time it takes to search for that element, that is $O(n)$ for an array of size n (swapping the element to the end takes constant time). Furthermore, the resizing operation that takes place during an "expensive" call of delete, is an extra $O(n)$ operation, thus deleting an arbitrary element takes $O(n)$ time. Since inserting an element takes $O(1)$ amortized time, over each $\%_{del}$ run, the time increases the time taken to execute all inserts is bounded by $O(n)$. Thus a single delete operations adds another $O(n)$ to the run time of the sequence, therefore increasing the number of delete operations will dominate run time of the sequence. Furthermore as we increase $\%_{del}$ the increase in run time should be at most proportional to the increase in $\%_{del}$. Since there are exactly 10^6 operations, there is going to be at most 10^6 elements in the array, thus each delete can only add at most a constant amount of time to the runtime of the sequence. From experiment 1, it was empirically derived that it would take on average 21ms complete 1M insertions, thus it can be deduced that the increase runtime is dominated by the deletions in this sequence. A linear regression can be fit, and a corresponding correlation coefficient of $R^2 = 0.9842$ is obtained, suggesting a linear trend in the experimental data, thus supporting our theoretical expectation. (*Note: linear regression was not shown on the graph, but data is supplied in the appendix*). If we consider the $0.1\%_{del}$ run, a possible reason why the dynamic array outperforms the treap in this instance is that the time taken to search 0.1% of the elements is shorter than that of the search time in the treap. This still follows with our theoretical expectation as we are expecting, asymptotically that the treap should outperform the dynamic array.

Experiment 3: Time vs $\%_{sch}$ (Search-Insertion Sequences)

Description

Insertion/deletion sequences were generated with percentage of deletion being varied each run, the time to execute each of the sequences was collected and then plotted.

Results

This experiment was run 5 times, on each run, new data was being generated and timed. The means of each runtime were then plotted.

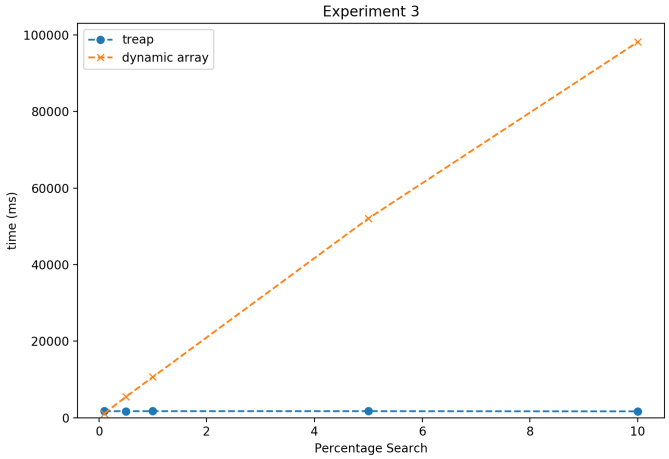

Figure 3: Experiment 3 Results

Analysis

Figure 3 shows that with the exception of the $0.1\%_{sch}$ run, the treap out performs the treap when across all insertion only runs. Much of the analysis of experiment two can be applied to explain the results of experiment 3. This is due to the fact that searching an array has the same complexity as deleting an arbitrary element (as deleting is dominated by the time to search), that is $O(n)$. The time to search a treap is equivalent to the time taken to insert (and similarly delete), thus for a treap the time taken to search an element is $O(\log d)$ if the treap contains d nodes. Thus it is expected that the treap should have a constant running time across a fixed sequence length of varying $\%_{sch}$. Figure 3 shows that the run time of the sequences executed on the treap appears to be relatively constant, taking 1717-1790ms to complete. As with experiment 2, as the number of $\%_{sch}$ increases in the sequence, the total run time is dominated by the time taken to search for an arbitrary element in the array as the total time taken to insert elements is bounded by the length of the sequence which is constant. As discussed in experiment 2, we expect that increasing the $\%_{sch}$ should yield at most a linear increase in sequence run time as adding an extra search operation to the sequence will at most a constant to the run time as the length of the array is bounded by 10^6. Figure 3 shows that our experimental results yielded an increase in run time over increasing $\%_{sch}$. Furthermore as with experimented 2, a linear regression was fitted over the data, and a linear correlation of $R^2 = 0.9992$ suggesting a strong linear correlation over the experimental data, thus supporting our theoretical expectation.

Experiment 4: Time vs Deletion-Search-Insertion Sequences

Description

Insertion/deletion sequences were generated with percentage of deletion being varied each run, the time to execute each of the sequences was collected and then plotted.

Results

This experiment was run 5 times, on each run, new data was being generated and timed. The means of each runtime were then plotted.

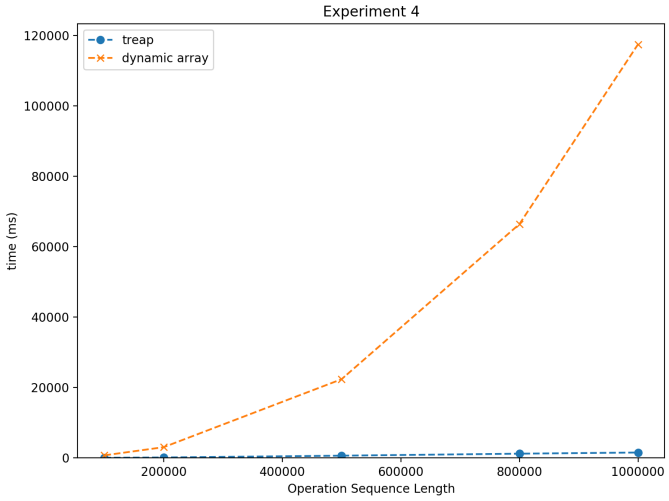

Figure 4: Experiment 4 Results

Analysis

Figure 4 shows that across all sequence lengths, the treap out performs the dynamic array. As determined in previous experiments, the expected run time of insertion into a randomised treap is $O(\log n)$, as is deletion and search operations. Thus as all three operations have the same expected run time we expect that given a sequence of length n operations, the runtime of the sequence is bounded by $n \log n$. The dynamic array on the other hand has different run times for each of the insertion, deletion and search operations. For insertion (using push-back) it is bounded by $O(1)$ amortized time. Deletion and searching an arbitrary element are both bounded by $O(n)$ time. As explained in the analysis of experiments 2 and 3 the time taken to delete and search for an arbitrary element will dominate the insertion time, thus if we consider the worst case time for search and delete, we can bound the run time of the sequence. If the probability that a search operation takes place is 0.05 and 0.05 for that of the deletion operation, we expect the number of search operations to occur to be $0.05n$ for each search and delete. As explained in the analysis of 2, adding a single delete operation to the sequence (or search) will increase the search time by at most $O(n)$ thus we can bound the time taken to execute all search or delete operations by at most $O(0.05n^2 + 0.05n^2)$. Thus for a dynamic array we expect that the time taken to execute this sequence is $O(n^2)$. *(Note: in the appendix an expected bound for this runtime of $O(n^2)$ is given is also given).* Hence, the treap should out perform the dynamic array over all sequence lengths. Figure 4 shows that the dynamic array seems to be increasing more than a proportional amount to

the operation sequence length and is greater than the run time of the treap at every sequence length. Thus our experimental findings correlate with that of our theoretical expectation.

Conclusion

The four experiments conducted evaluated the performance of the treap and dynamic array data structures under different cases. For insertion only sequences, the dynamic array outperformed the heap due to the efficiency of the $O(1)$ amortized time compared to the treaps $O(\log n)$ insertion time (n is the size of the treap). But for both mixed operation sequences of fixed length, and varying percentage of search operations or delete operations where the sequences length was fixed, the treap out performed the dynamic array the dynamic array in run time. This is because the treap was able to perform all three operations in $O(\log n)$ time whilst the dynamic array had linear time complexity of both delete and search which dominated the overall run time of the sequence of oeprations.

Appendix

Data

Experiment 1

TREAP

100000 : 68.37452ms
200000 : 170.73516ms
500000 : 636.4596ms
800000 : 1290.79444ms
1000000 : 1743.6106ms

DYNAMIC ARR

100000 : 2.01952ms
200000 : 4.72612ms
500000 : 10.5344ms
800000 : 18.86704ms
1000000 : 21.00956ms

Experiment 2

TREAP

0.1 : 1738.8ms
0.5 : 1727.6ms
1.0 : 1825.8ms
5.0 : 1718.2ms
10.0 : 1609.0ms

DYNAMIC ARR

0.1 : 598.8ms
0.5 : 2971.4ms
1.0 : 6371.4ms
5.0 : 44201.8ms
10.0 : 119787.0ms

Experiment 3

TREAP

0.1 : 1738.8ms
0.5 : 1727.6ms
1.0 : 1825.8ms
5.0 : 1718.2ms
10.0 : 1609.0ms

DYNAMIC ARR

0.1 : 598.8ms
0.5 : 2971.4ms
1.0 : 6371.4ms
5.0 : 44201.8ms
10.0 : 119787.0ms

Experiment 4

TREAP

100000 : 59.0ms
200000 : 139.8ms
500000 : 651.0ms
800000 : 1218.0ms
1000000 : 1546.0ms

DYNAMIC ARR

100000 : 745.0ms
200000 : 3062.4ms
500000 : 22369.4ms
800000 : 66441.6ms
1000000 : 117512.2ms

Bounding Dynamic Array Runtime for Experiment 4

Let S be the sequence of operations $\sigma_1, \sigma_2, ..., \sigma_n$ where each σ_i has the following probabilities. $\Pr[\sigma_i = \textbf{search}] = 0.05$, $\Pr[\sigma_i = \textbf{delete}] = 0.05$ and $\Pr[\sigma_i = \textbf{insert}] = 0.9$.
Let $T[\sigma_i]$ be the run time of operation i, thus we can define the following over an array of length n:
$T[\sigma_i = \textbf{delete}] = O(n) = an + b$, $T[\sigma_i = \textbf{search}] = O(n) = cn + d$,
$T[\sigma_i = \textbf{insert}] = O(1) = e$, where $n \in \mathbb{N} \cup \{0\}$, $a, c \in \mathbb{R}^+$ and $b, d, e \in \mathbb{R}$
Therefore we can find the expected runtime of S as follows:

$$E[T[S]] = E[T[\sigma_1] + T[\sigma_2] + ... + T[\sigma_n]]$$
$$E[T[S]] = \sum_{i=1}^{n} E[T[\sigma_i]] \text{ (By the linearity of expectation)}$$

If we have a sequence length of i, we can bound the size of the array at each step by i as we can have no more than i insertions, thus

$$\begin{aligned}
E[T[S]] = \sum_{i=1}^{n} \Big(&\Pr[\sigma_i = \textbf{search}] \times T[\sigma_i = \textbf{search}] \\
&+ \Pr[\sigma_i = \textbf{delete}] \times T[\sigma_i = \textbf{delete}] \\
&+ \Pr[\sigma_i = \textbf{insert}] \times T[\sigma_i = \textbf{insert}] \Big) \\
\leq \sum_{i=1}^{n} &\Big(0.05(ai + b) + 0.05(ci + d) + 0.9e \Big) \\
= \sum_{i=1}^{n} &\Big(i(0.05a + 0.05c) + (0.9e + 0.05b + 0.05d) \Big) \\
= &(0.05a + 0.05c) \sum_{i=1}^{n} i + n(0.9e + 0.05b + 0.05d) \\
= &(0.05a + 0.05c) \frac{n(n-1)}{2} + n(0.9e + 0.05b + 0.05d)
\end{aligned}$$

Thus showing that $E[T[S]] \in O(n^2)$