COMP90077 Project1 Report

Time Complexity Experimental Evaluation on Treap

Xiuge Chen 961392

1. Introduction

In computer science, the binary search tree (BST) is one of the typical data structures that utilize the idea of binary search. A BST is a tree where each node contains one key and two child nodes, where all the nodes on the left subtree have smaller keys, and nodes on the right subtree have larger keys. A BST supports search, insertion and deletion operations with time complexity *O(tree height)*, therefore ideally if the tree is balanced, all operations could be performed in *O(log n)*. However, in the worst case, the height of a tree could be *O(n)*, thus scientists have developed many variant BST data structures to help maintain the balance.

Treap was one of such self-balancing BST data structures which first was introduced by Seidel and Aragon in 1989 [1]. Briefly speaking, Treap is a randomized binary search tree that maintains its height by combining BST and min-heap with randomly generated priority, so that all the operations could be performed in *O(log n)* expected time.

In this experiment, Treap and a baseline data structure, dynamic array, are implemented and tested against various lengths of data and different percentages of different operations, in order to measure the effectiveness of Treap in practice.

2. Theoretical Background

2.1 Treap

According to [1], each node in Treap contains a key, a priority, and two child nodes, so it is both a BST and a min-heap. Below we present the definition of each operation in Treap and its time complexity.

1. Priority maintenance (x): do BST tree rotations on node x to rotate its level until the min-heap condition is satisfied. Since each rotation changes the node level by one, there are at most *O(tree height)* operations. Aragon [1] proved that in insertion and deletion, only constant *O(1)* number of rotations in expectation needs to be performed to maintain priority.
2. Insertion: Do a BST insertion to insert the new node into the appropriate position, then perform priority maintenance. So the time complexity of insertion will be *O(tree height + priority maintenance)*.
3. Deletion: Find the first matching node via BST search, increase the node priority to the maximum and perform priority maintenance until it is rotated to the leaf, delete this leaf. Thus deletion will cost *O(tree height + priority maintenance)* time to perform.
4. Search: Do a BST search until the first match or reach the leaf node. Therefore it has the same time complexity as the BST, *O(tree height)*.

As proved in [1], *O(tree height)* is expected to be *O(log n)* (but the worst-case is still *O(n)*), where n is the number of nodes.

2.2 Baseline: Dynamic Array

Here we choose the dynamic array as our baseline with each operation defined as follows.

1. Insertion: put the new elements at the next empty cell of the array, if the array is full, double its size first. The worst time complexity of insertion is O(n) since we need to copy all existing elements after size doubling, but as proved in [2], the amortized time-bound is just O(1).
2. Deletion: Search the elements, if found, swap the first matching with the last element, then delete the last element. If the result array has less than ¼ cells being filled, shrink its size to half. So the cost of deletion is O(search + shrink size) = O(n + n) = O(n).
3. Search: Check each cell of the array starting from the very beginning until we find the first match. The time complexity of the search is simply O(n).

	Treap	Baseline
Insertion Time	Expected: *O(log n)* Worst: *O(n)*	Amortized: *O(1)* Worst: *O(n)*
Deletion Time	Expected: *O(log n)* Worst: *O(n)*	*O(n)*
Search Time	Expected: *O(log n)* Worst: *O(n)*	*O(n)*

Table 2-1: Theoretical comparison between Treap and baseline.

3. Evaluation Metrics

3.1 Overall Running Time

To measure the actual time cost of each operation in practice, it is sufficient to measure the overall running time of a sequence of operations. Therefore, the start and end time (in milliseconds) of different experiments are recorded.

4. Experiment Preparation

4.1 Implementation

C++ implementation is chosen here for several reasons. Firstly, it is a very powerful middle-level language that could be used to build time-critical programs. Secondly, it is easier to program compared to C as it supports object-oriented programming. Lastly, it provides many useful built-in libraries that could help reduce the workload.

Specifically, besides all the details in section 2, we also include an id for each Treap node to help generate deletion data and break ties between the same keys. As for the dynamic array, we choose to develop it on the basis of the C++'s STL (standard template library) vector.

For better distribution, the random generator for priority assignment and data generation is implemented through the *uniform_*_distribution* functions from C++'s built-in random library. And the recording of the overall running time is implemented using C++'s chrono library. Spdlog library [3] is used for fast results logging.

4.2 Experiment Environment

We choose C++ 20 with CMake (version >= 3.15) and Clang compiler (version 11.0.0) to implement our data structures. All experiments are performed at a MacBook Pro, with macOS Catalina (10.15.3) operating system, 2 GHz Dual-Core Intel Core i5 processor and 8 GB 1867 MHz LPDDR3 memory.

4.3 Data Generation

Experiment data is artificially generated based on different purposes. Each data element contains two integers, a key, and a unique identifier id, as well as a type indicating insertion, deletion or selection. To make the id uniquely identify each new insertion, we keep track of the next available id, id_{next} , and let the initial id_{next} be 1. We also have two hashmaps *mGenInsert* and *mGenDelete* to store all insertions and deletions, where each map is in the form *(id, key)*. Let the universe of keys be $U=[0, 10^7]$, different operations are generated as follows.

1. Insertion: uniformly pick a key *k* from the universe *U*, assign id *i* to be id_{next} , increment id_{next} by one. Return a 3-tuple *(insert, k, i)*.
2. Deletion: randomly select an id *i* from [1, id_{next}-1], check *mGenDelete* if *i* is already deleted, if not, get the id-key pair *(i, k)* from *mGenInsert* and put it into mGenDelete. Otherwise, uniformly choose a deletion key *k* from the universe. Return a 3-tuple *(delete, k, 0)*.
3. Search: uniformly picked a key *k* from the universe *U*. Return a 3-tuple *(search, k, 0)*.

Data will be pre-generated and stored in a vector before starting experiments, so that we could accurately measure the runtime of each data structure. For each experiment, we used the same sequence of data to test both Treap and the baseline to ensure a fair comparison.

4.4 Experiment Settings

To compare the time complexity between Treap and the baseline, we performed the following four different experiments:

1. Time v.s. Insertion Number: perform different sequences of insertion with length 0.1M, 0.2M, 0.5M, 0.8M, 1M, where M stands for 10^6.
2. Time v.s. Deletion Percentage: Fixed the length to be 1M, allow insertion and deletion operations, perform different sequences with deletion proportion equals 0.1%, 0.5%, 1%, 5%, 10%.
3. Time v.s. Search Percentage: Fixed the length to be 1M, allow insertion and search operations, perform different sequences with search proportion equals 0.1%, 0.5%, 1%, 5%, 10%.

4. Time v.s. Mixed Operation Number: Mix all operations and fix the probability of deletion and search to be 5% respectively, perform different sequences with length 0.1M, 0.2M, 0.5M, 0.8M, 1M.

5. Result and Discussion

The results of each experiment *i* are shown in Table 5-*i* and Figure 5-*i*. For each table, each row represents one data structure and each column stands for one task of that experiment. Each cell indicates the overall running time of one data structure on one task, and we use bold font to mark the better one that has better time complexity.

5.1 Time v.s. Insertion Number

As shown in Table 5-1 and Figure 5-1, the results we obtained from experiment 1 align with our theoretical analysis about insertion. First of all, our baseline, dynamic array, is significantly faster than Treap for any length of insertion, so that it should have the lower insertion time-bound. Moreover, on the one hand, the overall running time of the dynamic array has a roughly linear relationship with the insertion length, thus its insertion time complexity should be constant or *O(1)*. On the other hand, Treap doesn't have such a perfect constant growth rate, adding the same length of insertions to a longer sequence will cost slightly more time. Such observations match with what we analyzed in section 2.1, the time complexity of Treap should be faster than *O(1)* but slower than *O(n)*, that is *O(log n)*. In practice, the amortized and expected bound of insertion could still be achieved.

Insertion Length	0.1M	0.2M	0.5M	0.8M	1M
Treap	220 ms	500 ms	1363 ms	2196 ms	2818 ms
Dynamic Array	**12 ms**	**31 ms**	**58 ms**	**96 ms**	**114 ms**

Table 5-1: Time complexity of different length of insertion

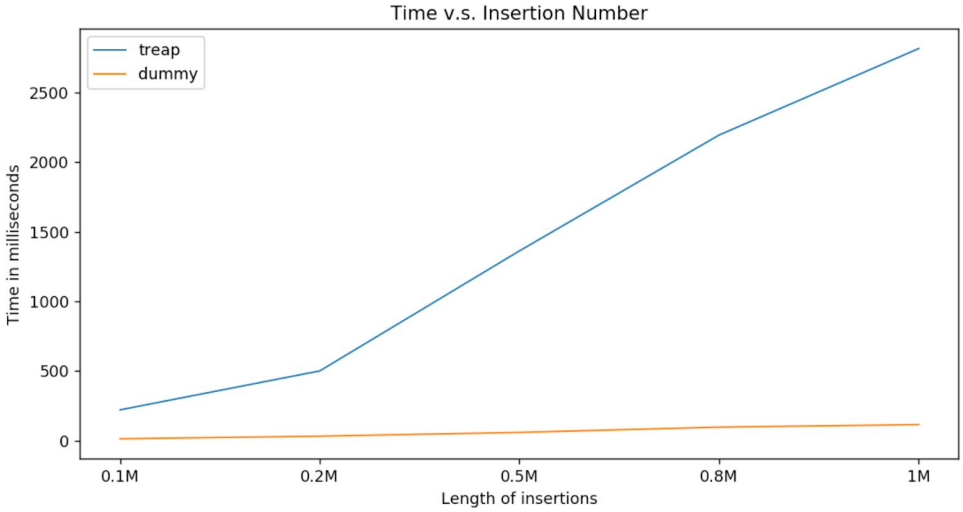

Figure 5-1: Time complexity of different length of insertion

5.2 Time v.s. Deletion Percentage

Table 5-2 and Figure 5-2 give the results when we fixed the sequence length and include different percentages of deletions. In our theoretical analysis, the dynamic array has larger deletion time complexity (*O(n)*) than Treap (expected *O(log n)*). But when we only include a very small proportion of deletion (0.1%), the dynamic array is still faster than Treap, since most of their operations are still insertions. As we increase the probability of deletion, the overall running time of the dynamic array grows rapidly and quickly overtakes Treap. The running time of Treap, however, is comparable more stable, it stays around the same even when we make the deletion percentage 100 times larger (from 0.1% to 10%). Besides, there is even a small decrease in Treap's running time as deletion percentage increased, since including more deletions will result in less Treap nodes, and correspondly the height of Treap will be reduced slightly (not a lot because of the log factor). Therefore, our theoretical analysis of deletion is again aligning with the experiment results. To have roughly the same time cost on all levels of deletion percentages, the deletion time complexity of Treap should be the same as its insertion time complexity, *O(log n)*. Thus this expected time-bound could also be achieved in practice. As for the dynamic array, it must have a larger deletion time bound to be able to surpass Treap very quickly, and based on the relative difference between dynamic array and Treap, our time bound about dynamic array deletion, *O(n)*, is correct.

Deletion Percentage	0.1%	0.5%	1%	5%	10%
Treap	2842 ms	**2828 ms**	**2866 ms**	**2738 ms**	**2684 ms**
Dynamic Array	**1279 ms**	9688 ms	22655 ms	101810 ms	198365 ms

Table 5-2: Time complexity of different percentage of deletion

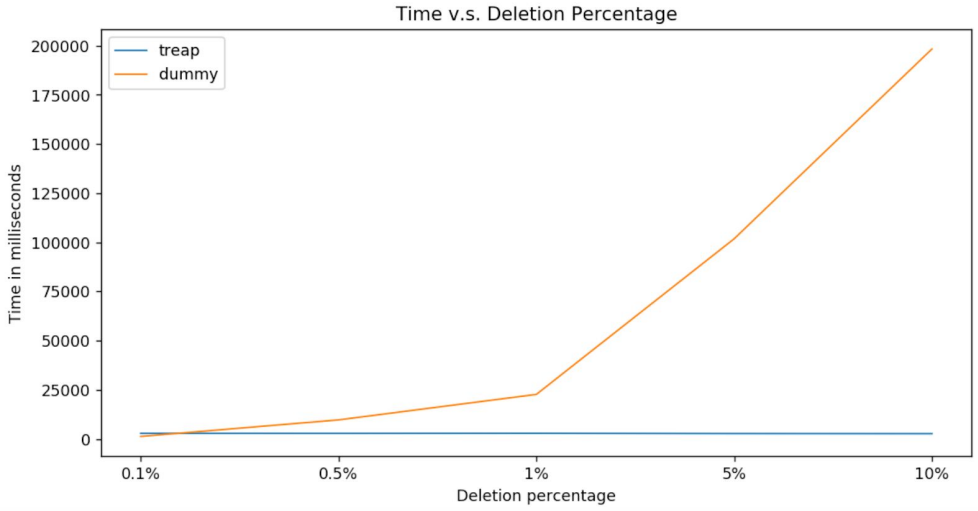

Figure 5-2: Time complexity of different percentage of deletion

5.3 Time v.s. Search Percentage

As represented by Table 5-3 and Figure 5-3, we fixed the number of operations and added different proportions of searches instead of deletions. The results of this experiment are very similar to the results in section 5.2, except that the Treap is the better choice even when only having a very small percentage of search (0.1%). Since both Treap and dynamic arrays have the same time complexity for search and deletion, so the results of this experiment should be similar to experiment 2. Not surprisingly, not only the actual time that Treap spent on each level is very similar to experiment 2, but also there is still a small decrease in running time when increasing the proportions because less insertion and nodes, and the log factor reduces its influence on tree height. This indicates that the search time complexity of Treap should be also the same as its insertion and deletion (*O(log n)*), and we could achieve this expectation bound in practice. The dynamic array has worse search time complexity, which matches its theoretical time bound *O(n)*. It is interesting that with the same percentage, experiment 3 (search) is much slower than experiment 2 (deletion), and the reasons are two-fold. Firstly, the result arrays are shorter in experiment 2 due to many deletion operations. Secondly, the different data generation process makes deletion keys more likely to exist in the data structures than the search keys, so that the search operation is more likely to touch the worst scenario (i.e. traversing through the whole array). However, here search in Treap only costs slightly more because the log factor again reduces the impact of different data lengths, in another word, the resulting tree heights in experiments 2 and 3 are still very close.

Search Percentage	0.1%	0.5%	1%	5%	10%
Treap	**2848 ms**	**2803 ms**	**2796 ms**	**2731 ms**	**2740 ms**
Dynamic Array	6208 ms	30616 ms	61071 ms	288776 ms	548543 ms

Table 5-3: Time complexity of different percentage of search

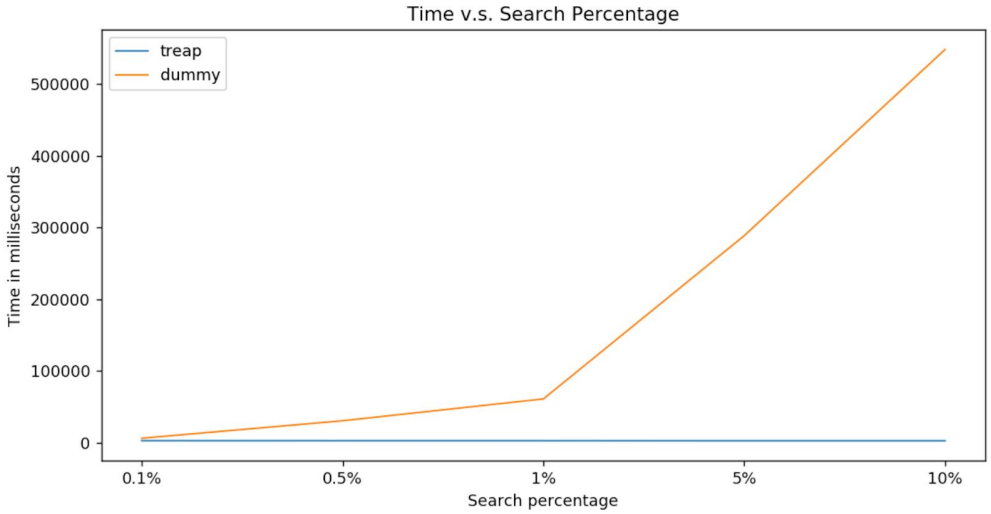

Figure 5-3: Time complexity of different percentage of search

5.4 Time v.s. Mixed Operation Number

In the last experiment, we allow all operations with fixed probability and vary the length of the sequences, the results are displayed in Table 5-4 and Figure 5-4. Theoretically, all operations of Treap have the same time complexity O(log n), thus the overall running time of experiments 1 and 4 should be pretty close. Similarly, the dynamic array has the same time bound for search and deletion, thus the 1M case of this experiment could be approximately considered as the combination of the 5% case of experiments 2 and 3. And our results below align with these predictions as well, the overall running time of Treap is very similar to experiment 1, and most levels have even smaller running time because the fewer insertions and fewer nodes, thus including more deletion/search in Treap actually will help slightly decrease its overall running time. But the dynamic array will be hugely negatively influenced by larger percentages of deletion/search, as the running time of deletion and search is *O(n)*. So for the same sequence length, the running time of experiment 4 is significantly larger than experiment 1.

Length of Mixed Operations	0.1M	0.2M	0.5M	0.8M	1M
Treap	**246 ms**	**410 ms**	**1181 ms**	**2057 ms**	**2663 ms**
Dynamic Array	3275 ms	14151 ms	87335 ms	231784 ms	363861 ms

Table 5-4: Time complexity of different length of mixed operations

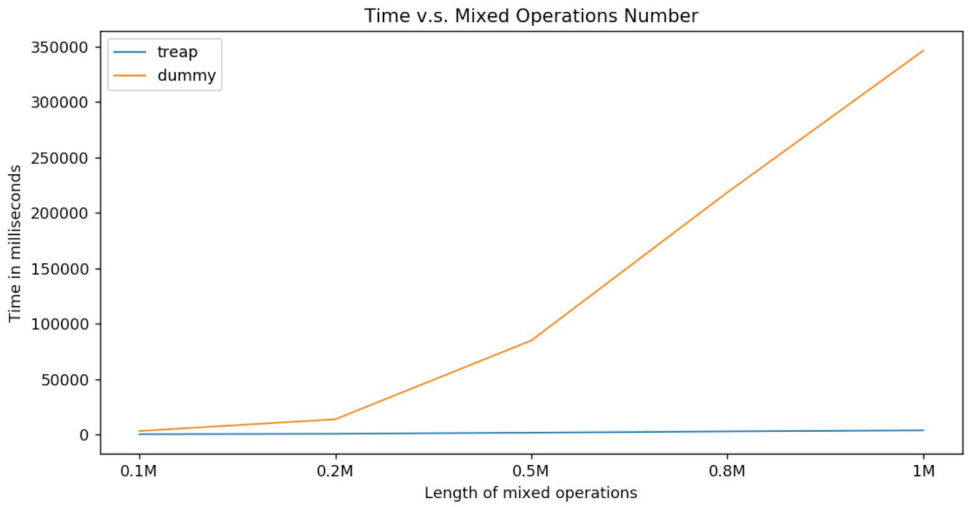

Figure 5-4: Time complexity of different length of mixed operations

6. Conclusion

In conclusion, all of the theoretical expected and amortized time bound could be successfully achieved in practice.

On the one hand, our baseline data structure, dynamic array, is better at insertion operations since it has amortized time bound of O(1) for insertion, while the insertion of Treap is expected to be done in O(log n) time. Therefore, if the probability of having operations other than insertion is relatively low (under 0.1%), the dynamic array will have better performance.

On the other hand, the dynamic array has O(n) time bound for both deletion and search, but it only costs Treap O(log n) time in expectation. Consequently, for a sequence of mixed operations where delete/search operations appear frequently (higher than 0.5%), using Treap could significantly reduce the overall time cost.

It is not surprising that including the search will cost more time than adding the same percentage of deletion in the dynamic array since deletion helps reduce the array size. The way we generate data also makes deletion more likely to exist in the data structure than search, resulting in more search time. But this does not quite hold for the Treap, as the log factor surpasses such influence, the resulting Treap heights are very close. Moreover, since all operations of Treap share the same time complexity O(log n), for a length fixed sequence, Treap has another advantage that increasing percentages of deletion and search will even help reduce the overall running time, as there will be fewer insertions and the Treap height will become slightly smaller.

7. References

[1] Aragon, Cecilia R., and Raimund G. Seidel. "Randomized search trees." (1989): 540-545.
[2] Junhao, G. and Wirth, T. (n.d.). *Week 2 Slides, COMP90077*. [online] Melbourne: University of Melbourne. Available at: https://canvas.lms.unimelb.edu.au/courses/80169/files/2782958?module_item_id=1794623.
[3] Melman Gabi, spdlog, (2020), GitHub repository, https://github.com/gabime/spdlog.