COMP90077 - Assignment 1
An Experimental Study on Randomized Treaps

Tanzid Sultan (ID# 1430660)

1 Introduction

Many algorithms of fundamental importance require the maintenance of a dynamically changing set of ordered items. Therefore there is a need for data structures that can represent such sets of items and support efficient searching, insertion and deletion of items from the set. An important class of data structures for solving this problem relies on the use of **ordered binary trees**. In particular, there are two types of tree orderings that are of great interest: *symmetric-ordering* and *heap-ordering*. **Balanced binary search trees** (BST) such as Red-Black and AVL trees, are symmetric-ordered binary trees that utilize explicit *balancing operations* that make local changes to the tree structure in order to maintain minimal height for the tree (which for a tree containing n items is $O(\log_2 n)$). Since the theoretical worst case running-time of the access, insertion and deletion operations into a BST are upper-bounded by the height of the tree, these balanced trees are therefore able to achieve good theoretical performance. On the other hand, **binary min-heaps** are heap-ordered binary trees that are primarily designed for efficiently finding items that have a minimum *priority*, updating priorities of existing items and insertions and deletions of items. However it is not possible to efficiently perform the search operation for arbitrary items in a min-heap.

A **treap** is a binary tree which combines both symmetric-ordering and heap-ordering. Each item has two (real number) attributes: a *key* and a *priority*. The treap is symmetric-ordered with respect to the keys and heap-ordered with respect to the priorities, leading to a unique structure of the tree for any particular key-priority assignments on a set of items. The main utility of treaps comes from the use of **randomization**. By assigning priorities to a set of items uniformly at random, the resulting treap is **expected**[1] to have the minimum possible height, i.e. $O(\log_2 n)$. Therefore such a treap is able to achieve good theoretical performance, without the use of any complicated explicit balancing operations. Simply maintaining both the BST/symmetric-order and heap-order properties at all times suffices to ensure minimal expected height. Therefore we can think of the treap as achieving balance indirectly via randomization, which may allow treaps to be a viable and simpler alternative to explicitly-balanced binary search tree data structures for many applications.

The goal of this project is to perform a series of experiments in which we study and analyze the practical performance of a randomized treap implementation. By doing so, we wish to definitively

[1] By "expected height", we mean that if we generate an ensemble of n randomized treaps on the same set of items (and always randomly chosen priorities), then the average height of all treaps in the ensemble converges towards the expected height as $n \to \infty$.

find out how well the performance under practical setting matches with theoretical expectations which can then allow us to draw conclusions about the utility of treaps for practical applications. When performing such experimental studies, it is useful to have a baseline model for comparison, which we shall call the **Competitor**. In our case, we shall use a simple **Dynamic Array** data structure as the Competitor as it's properties are robust and simple to analyze.

2 Experiment Environment

The experiments have been conducted on a personal laptop equipped with a *13th generation Intel Core-i7* processor and running the *Microsoft Windows 11 Home* operating system. The programs were executed within a Windows Subsystem for Linux (WSL 2) session which is a virtual/containerized Linux environment that provides native Linux compatibility and performance. The following table provides some pertinent information about the system hardware:

Specifications	
Processor Number	i7-13700HX
Word-Size/Instruction Set	64 bit
Physical Cores (Threads)	16 (24)
Frequency Range	2.1 - 5.0 GHz
L1 Cache	1.4 MB
L2 Cache	14 MB
L3 Cache	30 MB
Physical Memory (RAM)	32 GB
RAM speed	4800 MHz

Table 1: CPU and memory specifications

We have chosen to implement our programs entirely using the **C++** language, primarily due to its high performance capability and efficient memory management. In addition, C++ is an object-oriented programming language which allowed us to neatly encapsulate our data generator, treap and dynamic array implementations inside separate object classes leading to a clean and organised code-base. The programs have been compiled using the **g++** compiler (C++17 standard) obtained from the *GNU Compiler Collection v11.4*. We chose not to enable any special compiler optimization features, due to simplicity reasons and to prevent the possibility of such features to introduce hidden effects that may interfere with our algorithm analysis. The compiled programs were executed directly from the bash command line in the WSL 2 environment. Program execution times for our experiments were measured using the *high_resolution_clock* class provided by the C++ standard *chrono* library, which is known to have high precision. No additional software libraries beyond those already provided by the C++ standard library were utilized.

In terms of memory usage, we note that the cache and physical memory size are adequate and substantial which eliminated any risk of memory overflow or memory-related bottleneck issues when running our experiments. In particular, the memory consumption of our data structures during any given experiment is typically on the order of a few tens of megabytes which is small enough to fit across the L2 and L3 caches and consequently there was no significant slowdown due to memory access.

3 Data Generation

To test the performance of our treap implementation and perform the necessary experiments, we have implemented a **data generator** class which serves two main purposes: (i) to generate data elements and (ii) to generate operations such as insertion, deletion and search. Listing 1. shows the header file for our data generator implementation in C++.

```
#ifndef DATA_GENERATOR_H
#define DATA_GENERATOR_H
#include <vector>
#include <random>

class DataGenerator {
    private:
        int id_next;
        std::vector<std::vector<int>> data; // vector for storing generated data
        std::mt19937 generator;  // pseudo-rng (Mersenne Twister)
        std::uniform_real_distribution<double> distribution;
        static int SEED;
        static float ZERO;

        int generateRandInt(int a, int b);
        float generateRand();

    public:
        DataGenerator();  // constructor
        ~DataGenerator(); // destructor

        std::vector<int> genElement();
        std::vector<int> genInsertion();
        std::vector<int> genDeletion();
        std::vector<int> genSearch();
        std::vector<std::vector<int>> genOpSequence(int L, float del_percent, float sch_percent);
};

#endif
```

Listing 1: Data Generator Interface

Once instantiated, a data generator object internally maintains a state which contains the following: (i) an integer variable *id_next* (initialized to 1) which will be assigned to the next data element to be generated and incremented by 1 each time, (ii) a dynamic array storing all data elements that have been generated so far (indexed by element *id*) and (iii) a pseudo-random number generator[2]. The data generator also provides several interfaces that will be used by our client program that runs the experiments. These interfaces are described is detail below:

- *genElement():* This creates and returns a new data element which is a tuple of the form (id, key). The integer *id* is a unique identifier for the element which is set to be the next id which is maintained in the internal state of the data generator. Because the next id is incremented by

[2] We use the 32-bit *Mersenne Twister* random-number generator (rng) provided by the C++ standard library. This is a general-purpose rng that is widely regarded to be efficient, robust and suitable for almost all practical applications including cryptography.

1 after creating each new data element, this ensures no two elements will ever have the same id. The key is an integer drawn uniformly at random from the range $[0, 10^7]$ using the pseudo-random number generator. A copy of the data element is also appended into the dynamic array that stores all data elements that have been generated so far.

- *genInsertion():* This generates an insertion operation which is a tuple of the form $(1, x)$, where x is a new data element created by calling the $genElement()$ interface. The 1 inside the tuple is used to indicate that this tuple represents an insertion operation. This tuple can then be provided to an appropriate data structure, in our case a treap or competitor, which will then use the 1 in the tuple to identify this operation as an insertion and then execute an insertion taking x as the input.

- *genDeletion():* This generates a delete operation which is a tuple of the form $(2, key_{del})$. First, an integer id is drawn uniformly at random from the range $[1, id_{next} - 1]$ (which is the range over *ids* of all data elements which have been generated so far) using the pseudo-random number generator. Since all data elements generated so far are currently stored inside a dynamic array which is part of the internal state of the data generator, the element x corresponding to this randomly selected id is retreived from the array and then key_{del} is set to be equal to $x.key$. The tuple $(2, key_{del})$ can then be provided to a treap or competitor, which will then use the 2 in the tuple to identify this operation as representing a deletion and then execute a deletion taking key_{del} as the input.

- *genSearch():* This generates a search operation which is a tuple of the form $(3, key_{sch})$. An integer key_{sch} is drawn uniformly at random from the range $[0, 10^7]$ using the pseudo-random number generator. The tuple $(3, key_{sch})$ can then be provided to a treap or competitor, which will then use the 3 in the tuple to identify this operation as representing a search and then execute a search taking key_{sch} as the input.

- $genOpSequence(L, \%_{del}, \%_{sch})$: This interface generates a sequence of operations of length L. Each operation in the sequence is drawn randomly such that with $\%_{del}$ probability a deletion is selected by calling $genDeletion()$, with $\%_{sch}$ probability a search is selected by calling $genSearch()$ and with the remaining $100 - \%_{del} - \%_{sch}$ probability, an insertion is selected by calling $genInsert()$. More specifically, each time, we draw a uniform random number p from the continuous range $[0, 1)$ using the random-number generator. If $p < \frac{\%_{del}}{100}$ we select a deletion, if $p < \frac{\%_{del} + \%_{sch}}{100}$ we select a search, otherwise we select an insertion. Therefore, $\%_{del}$, $\%_{sch}$ and $100 - \%_{del} - \%_{sch}$ are in-expectation percentages of deletion, search and insertion operations respectively in the sequence.

Each of the experiments that we have performed involved executing a particular sequence of operations σ on the treap and dynamic array data structures. The σ for an experiment is obtained by calling the $genOpSequence(L, \%_{del}, \%_{sch})$ interface of a data generator object with a given set of parameters $(L, \%_{del}, \%_{sch})$ pertaining to the experiment.

4 Data Structure Design

In this section, we provide some pertinent details about the implementation of our data structures which is intended to aid the reader in reproducing our experiments if they choose to do so.

4.1 Treap

Listing 2 shows the C++ header file which outlines the design structure of our treap implementation.

```cpp
#ifndef TREAP_H
#define TREAP_H
#include <vector>
#include <random>
#include <fstream>
#include <string>

// Treap node struct
struct Node {
    int id;
    int key;
    float priority;
    Node* parent;
    Node* left;
    Node* right;
};

// Treap Interface
class Treap {
    private:
        Node* root;
        int size;
        std::mt19937 generator;  // pseudo-rng (Mersenne Twister)
        std::uniform_real_distribution<double> distribution;  // continuous uniform distribution
        static int SEED;
        float generateRand();
        Node* bstInsert(Node* new_node, int id, int key, float priority);
        void maintainHeapPropertyInsert(Node* node);
        void rotateToLeaf(Node* node);
        void leftRotation(Node* node);
        void rightRotation(Node* node);
        void destroyTreap(Node* node);
        void exportToDot(Node* node, std::ofstream& dotFile);

    public:
        Treap(); // constructor
        ~Treap(); // destructor
        int getSize() const;
        void exportTree(const std::string& filename);
        int findTreeHeight() const;

        Node* searchItem(int key_sch) const;
        void  insertItem(int id, int key);
        void  deleteItem(int key_del);
        void  performOperation(std::vector<int> op);
};
#endif
```

Listing 2: Treap Interface

We have defined a composite data type (i.e a struct) for representing the nodes of the tree. Each node has attributes including *id*, *key*, *priority* in addition to pointers to it's parent and children.

The *Treap* class encapsulates our implementation of the treap data structure. The data structure is equipped with it's own pseudo-random number generator which is used to assign a uniform randomly sampled priority from $[0, 1)$ to each newly inserted nodes. The class contains a set of methods which include the interfaces for the insertion, deletion and search operations as well as various helper methods for facilitating these three main operations.

Helper methods such as *leftRotate()* and *rightRotate()* are used for maintenance of heap property violation that may occur during an insertion or deletion. Other helper methods such as *findTreeHeight()* can be used to compute the height of the treap (using bredth-first search), this is useful if we are interested in comparing the actual tree height with the theoretically expected height, although it is not needed for performing any of the three main operations. We have also included a helper method called *exportTree()* which generates a *.dot* file representation of the treap. This *.dot* file can then be utilized by the *Graphviz* software, available for most Linux distributions, to create a visualization of the binary tree[3]. We have found this visualization tool to be extremely useful for debugging purposes, specially when verifying whether the main operations have been implemented correctly, in particular to check the sequences of rotation operations involved during maintenance of the heap-property.

The client program which runs the experiments will primarily interact with an instantiated Treap object via the interfaces for the main operations. We briefly describe these interfaces below:

- *searchItem(key_sch):* This operation performs a binary search, i.e. in-order traversal starting from the root node, to find the a node x in the treap which has $x.key = key_{sch}$. if there are multiple nodes in the treap which have this key, then the operation returns the first node it finds. If such a node is not found, a *null* pointer is returned.

- *insertItem(id, key):* This operation first creates a new node with the given *id* and *key* and also assigns a priority value chosen uniformly at random from $[0, 1)$. It then performs in-order traversal to find an appropriate leaf position and inserts the new node at that position. If an existing node with the same key is encountered during the in-order traversal, then the id of that node is used to break the tie, in other words if the id of the new node is smaller than the id of the existing node, then the in-order traversal goes down the left path and vice versa. After insertion into the leaf position, we check for heap property violation and perform maintenance via rotations if needed.

- *deleteItem(key_del):* This operation performs a binary search to find a node x which has $x.key = key_{del}$. If such a node is found, it is then rotated down to a leaf position and removed from the tree.

- *performOperation(op):* Recall that the data generator provides interfaces for generating insertion, deletion and search operations. This operation is represented by a tuple which is of the form (i, x). The first element of the tuple i is an integer which equals 1 if the operation is an insertion, 2 for deletion and 3 for search. The second element of the tuple is the input argument corresponding to that operation, e.g. for an insertion $x = (id, key)$, for a deletion $x = key_{del}$, etc. Then this $performOperation(op)$ interface of the treap takes such an operation tuple $op = (i, x)$ as input and then executes the operation represented by this tuple.

[3] e.g. a png image of the tree can be created by running the command: `dot -Tpng input.dot -o output.png`

4.2 Dynamic Array

Listing 3 shows the C++ header file which outlines the design structure of our dynamic array implementation.

```
#ifndef DYNAMIC_ARRAY_H
#define DYNAMIC_ARRAY_H
#include <vector>

// Dynamic Array Interface
class DynamicArray {
    private:
        int capacity;
        int size;
        int** arr;

        void grow();
        void shrink();

    public:
        DynamicArray(); // constructor
        ~DynamicArray(); // destructor

        int getSize() const;
        int getCapacity() const;
        int* get(int idx) const;

        int* searchItem(int key_sch) const;
        void insertItem(int id, int key);
        void deleteItem(int key_del);
        void performOperation(std::vector<int> op);
};
#endif
```

Listing 3: Treap Interface

The *DynamicArray* class encapsulates our implementation of the dynamic array data structure. This class contains a similar set of methods as the Treap class, which include the interfaces for the insertion, deletion and search operations as well as some helper methods for facilitating these three main operations. Upon instantiating an object of this class, it contains an empty 2-dimensional array of capacity 2 (which is dynamically allocated via the C++ *new* operator) and maintains attributes for keeping track of the number of elements occupying the array and the array capacity. The array is 2-dimensional because it needs to store an $(\mathit{id}, \mathit{key})$ pair for each element. Resizing of the array may occur during insertions and deletions, which is facilitated by the *grow()* and *shrink()* helper methods. The client program which runs the experiments primarily interacts with an instantiated DynamicArray object via the interfaces for the main operations. We describe these interfaces below:

- *searchItem(key_sch):* This operation scans each element in the array sequentially from the beginning and returns the first element $x = (\mathit{id}, \mathit{key})$ for which $x.\mathit{key} = \mathit{key}_{sch}$. If such an element is not found, a *null* pointer is returned.

- *insertItem(id, key):* This operation inserts a new element $x = (\mathit{id}, \mathit{key})$ in the first empty slot in the array. It then check to see if the array is full. If the array is full, then a new array is created which has double the capacity, all elements from the original array are copied over to this new array and then the original array is discarded and replaced with this new array.

- *deleteItem(key_del):* This operation performs a search for an element $x = (id, key)$ which has $x.key = key_{del}$. If such an element is found, it is then swapped with the last element in the array and then removed. It then check to see if the number of elements in the array is less than 1/4 of the capacity. If so, then a new array is created which has half the capacity, all elements from the original array are copied over to this new array and then the original array is discarded and replaced with this new array.

- *performOperation(op):* This interface takes an operation tuple $op = (i, x)$ as input and then executes the operation represented by this tuple, same as the *performOperation(op)* interface of the Treap.

5 Experiments

Now we shall present and analyze the results from 4 different experiments which we have carried out. In each experiment, we have generated 5 different sequences of operations: $\sigma_1, \sigma_2, ..., \sigma_5$, then we ran each sequence of operations on both the treap and the competitor and measured their respective running times. At the beginning of each operation sequence, both the treap and competitor data structures are empty, i.e. they contain zero data elements initially. By performing these experiments, we aim to get a better understanding for how each data structure performs under a wide range of scenarios, e.g. how each data structure performs under varying sizes of the operation sequence or under varying relative proportions of the different types of operations in the sequence.

Before we go into discussing about each experiment, we will quickly summarize and compare the theoretical per-operation performance of the treap and dynamic array data structures in the table below.

Operation	Treap	Competitor
Insertion	$O(\log N)$ expected	$O(1)$ amortized
Deletion	$O(\log N)$ expected	$O(N)$ worst-case
Search	$O(\log N)$ expected	$O(N)$ worst-case

Table 2: Theoretical performance of treap vs dynamic array. N denotes the total number of elements contained in the data structure.

Note that the time complexity for all three operations for the treap is in $O(\log N)$ in expectation. For the dynamic array, insertion has an amortized cost in $O(1)$ while the deletion and search operation has worst-case cost in $O(N)$ (deletion of an item involves first finding the location of that item, and therefore has the same worst-case cost as a search). We have implicitly assumed a Word-RAM model of computation in which all arithmetic operations, logical operations and memory accesses require constant time. It is also worth noting that the theoretical space consumption of both data structures is bounded by $O(N)$. Finally, it should also be noted that each call to any random number generator used in our code has $O(1)$ cost.

5.1 Experiment 1

In our first experiment, we have generated 5 **insertion-only** sequences of varying lengths. The sequence lengths are chosen as follows: $0.1M$, $0.2M$, $0.5M$, $0.8M$ and $1M$ where M stands for 10^6. These sequences are generated by calling the $genOpSequence(L, \%del, \%sch)$ interface provided by our data generator with the arguments specified appropriately, i.e. we set $\%_{del} = 0$, $\%_{sch} = 0$ and L to one of the 5 corresponding values. After generating these 5 insertion sequences, each sequence is then executed separately on both the treap and competitor (both data structures are empty at the start of each sequence) and their running times are measured. Figure 1 shows a plot of the measured total running time vs insertion sequence length for both the treap and competitor. To mitigate some of the noise in the measurements, we have repeated the experiment up to 8 times for each insertion sequence. The points on the plot denote the average running time across all 8 trials and the error bars denote the standard deviation.

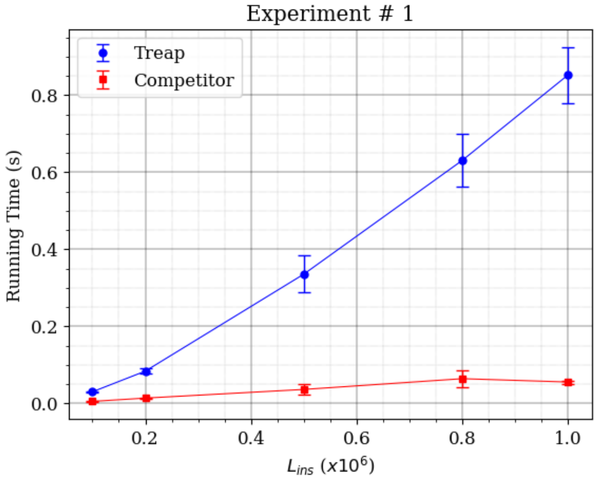


Figure 1: Time vs. Number of Insertions.

Measured running times for both the treap and the competitor increases monotonically, however it is clear that the running time for the treap grows faster as the sequence length increases. To understand this result, we will perform some mathematical analysis to estimate how the theoretical running times for an arbitrary insertion sequence scales with the sequence length. Let L be the total number of insertions in the sequence and let c_i denote the cost of the *ith* insertion. Here, we use the word *cost* interchangeably with *running time*. Then the total cost of the sequence is given by:

$$C = \sum_{i=1}^{L} c_i \tag{1}$$

First, let's consider the treap for which the expected cost of an insert operation is $E[c_i] \in O(\log n)$. Then the expected total cost of an insertion sequence is given by:

$$E[C] = E\left[\sum_{i=1}^{L} c_i\right] = \sum_{i=1}^{L} E[c_i] \in \sum_{i=1}^{L} O(\log n_i) \in O\left(\sum_{i=1}^{L} \log n_i\right) \tag{2}$$

where we have used the linearity property of the expectation and defined n_i to be the number of elements in the treap during the *ith* insertion. Since we perform an insertion-only sequence of operations and the treap is initially empty, we have $n_i = i$. Therefore we can write the following:

$$\sum_{i=1}^{L} \log n_i = \sum_{i=1}^{L} \log i = \log\left(\prod_{i}^{L} i\right) = \log(L!)$$

For large L, we can use *Stirling's formula* which gives us a useful approximation for the factorial: $L! \approx (\frac{L}{e})^L \sqrt{2\pi L} \Rightarrow \log(L!) \approx (L + \frac{1}{2}) \log L + \frac{1}{2} \log(2\pi) \in O(L \log L)$. Therefore we have the following upper-bound for the expected total cost of an insertion sequence into the treap:

$$E[C] \in O\left(\sum_{i=1}^{L} \log n_i\right) \in O(L \log L) \tag{3}$$

Next, we estimate the total cost of the insertion sequence for the competitor. Since the cost of a single insertion into a dynamic array is in $O(1)$ amortized, which means $c_i \in O(1)$ amortized, then we have the following upper-bound on the total cost:

$$C = \sum_{i=1}^{L} c_i \in \sum_{i=1}^{L} O(1) \in O(L) \tag{4}$$

Thus according to this mathematical analysis, the $O(L \log L)$ estimated theoretical total expected cost for the treap grows faster than the the $O(L)$ cost of the competitor as the insertion sequence length increases. In other words, the competitor will outperform the treap on the insertion-only sequence as sequence length gets larger. The results in Figure 1 indeed confirms this analysis and we also note that the shape of the blue curve, corresponding to the treap, indicates slightly faster than linear growth.

5.2 Experiment 2

In our second experiment, we allow the operation sequences to contain both insertions and deletions while keeping the sequence length fixed at $L = 1M$. We generate 5 different sequences with varying in-expectation percentages $\%_{del}$ of deletions, in particular we have chosen the values $\%_{del} = 0.1\%, 0.5\%, 1\%, 5\%, 10\%$. These sequences are generated by calling the $genOpSequence(L, \%del, \%sch)$ interface provided by our data generator with the arguments specified appropriately, i.e. we set $L = 1M$, $\%_{sch} = 0$ and $\%_{del}$ to one of the 5 corresponding values. Each sequence is then executed separately on both the treap and competitor. Just like in our previous experiment, we perform multiple trials and report the average measured total running times in Figure 2. Figure 2(a) (on the left) contains two curves, one for the treap and the other for the competitor. Due to the large discrepancy

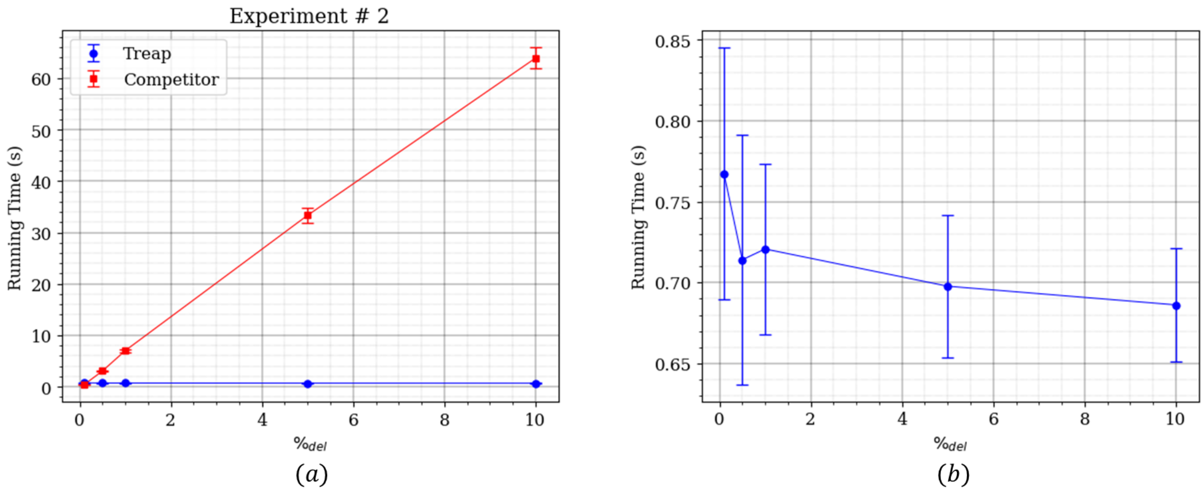


Figure 2: Time vs. Deletion Percentage: (a) Treap and Competitor (b) Treap only

between the treap and competitor running times, the running times for the treap seem almost negligible and are difficult to see in the plot. We have therefore provided a second plot which is Figure 2(b) (on the right) showing a close-up view of the treap curve only.

Looking at Figure 2(a), we immediately see that the measured running time for the competitor grows quite rapidly as $\%_{del}$ increases, while Figure 2(b) indicates that the measured running time for the treap gradually decreases. It is clear that the treap vastly outperforms the competitor, with the treap being roughly 100x faster when $\%del = 10$. Intuitively, these results make sense. For the dynamic array, the deletion operation is much more expensive than the insertions. In fact, each deletion on the dynamic array is expected to be much more expensive than a corresponding deletion on the treap. As $\%_{del}$ increases, the expected number of deletion operations in the sequence also increases, while the expected number of insertions decreases. This is why we observe a rapid increase in the total running time of the dynamic array as $\%_{del}$ increases. However, for the treap we observe a decrease in the total running time. This can be explained by the fact that a larger number of deletions means that the treap will contain fewer elements and therefore the expected height of the treap should be smaller, which leads to lower expected total running time. To confirm these intuitions, we can carry out some mathematical analysis to estimate upper-bounds on the theoretical performance of each data structure and find out how these upper-bounds scale with $\%_{del}$.

Let o_i be an independent random variable denoting the ith operation in a sequence of length L. Then we define p as the probability of the ith operation being a deletion, i.e. $p = Pr[o_i = deletion] = \frac{\%_{del}}{100}$. Since the sequence only consists of insertions and deletions, this means the probability of the ith operation being an insertion must be $Pr[o_i = insertion] = 1 - Pr[o_i = deletion] = 1 - p$. So, given that the actual cost of the ith operation is $c_i = cost(o_i)$, the expected cost of the ith operation must be the following:

$$\begin{aligned} E[c_i] &= Pr[o_i = insertion] \cdot cost(o_i = insertion) + Pr[o_i = deletion] \cdot cost(o_i = deletion) \\ &= (1 - p) \cdot cost(insertion) + p \cdot cost(deletion) \qquad (5) \end{aligned}$$

Note that this expectation is over the random variable o_i. This means the total cost of the operation sequence in expectation is given by:

$$
\begin{aligned}
E[C] &= \sum_{i=1}^{L} E[c_i] \\
&= (1-p)\sum_{i=1}^{L} cost(insertion) + p\sum_{i=1}^{L} cost(deletion) \qquad (6)
\end{aligned}
$$

For a treap, we know that the expected cost[4] for the ith operation is the same for both insertion and deletion, i.e. $cost(insertion) = cost(deletion) \in O(\log n_i)$, where n_i is the number of elements in the treap during the operation. Therefore we have the following for the expected total cost for the treap:

$$
E[C] \in O\left(\sum_{i=1}^{L} \log n_i\right) \tag{7}
$$

Since we now have both insertions and deletions, the number of elements in the data structure during the ith operation $n_i \neq i$. We can define a random variable X_k which equals 1 if the kth operation is an insertion and equals -1 if the kth operation is a deletion. So the probability $Pr[X_k = 1]$ is the same as $Pr[o_k = insertion]$ and similarly $Pr[X_k = -1] = Pr[o_k = deletion]$. Then we can express the total number of elements in the data structure after the ith operation as follows:

$$
n_i = \sum_{k=1}^{i} X_k
$$

Then the expected total number of elements in the data structure following the ith operation is given by:

$$
E[n_i] = \sum_{k=1}^{i} E[X_k] = \sum_{k=1}^{i} Pr[X_k = 1] - Pr[X_k = -1] = \sum_{k=1}^{i}((1-p)-p) = (1-2p)i \tag{8}
$$

When i is large, we can use the approximation $n_i \approx E[n_i] = (1-2p)i$. Substituting this into Equation (7) gives us the following expected total cost for the treap:

$$
E[C] \in O\left(\sum_{i=1}^{L} \log\left((1-2p)i\right)\right) = O\left(\sum_{i=1}^{L}(\log(1-2p) + log(i))\right) \approx O\Big(L\log(1-2p) + L\log(L)\Big) \tag{9}
$$

where we've used Striling's approximation on the last step, i.e.$\sum_{i=1}^{L} log(i) \approx L\log L$. Given L is a fixed constant, we have an $O(\log(1-2p))$ estimated theoretical upper-bound on the total cost, which is a decreasing function of p, which means the expected total cost of the operation sequence should decrease as $\%_{del}$ increases. This is indeed consistent with our experimental results, in particular, we observed from Figure 2(b) that the measured running time for the treap slowly decreases.

[4] To avoid confusion, we note that this *expected* cost of a treap operation is completely different from the expectation invoked in Equation (5), which is an expectation over the random variable o_i. Also see footnote in Page 1.

Next, we consider the worst-case cost of the *ith* operation for the dynamic array, which is given by the following:

$$c_i = cost(o_i) = \begin{cases} O(1) & \text{if } o_i = insertion \\ O(n_i) & \text{if } o_i = deletion \end{cases} \tag{10}$$

where n_i is the number of elements in the dynamic array during the operation. Then the total expected cost for the entire sequence of operations is given by:

$$\begin{aligned} E[C] &= (1-p)\sum_{i=1}^{L} cost(insertion) + p\sum_{i=1}^{L} cost(deletion) \\ &\in (1-p)\sum_{i=1}^{L} O(1) + p\sum_{i=1}^{L} O(n_i) \end{aligned}$$

Once again, we can invoke the approximation $n_i \approx E[n_i] = (1-2p)i$ to get the following:

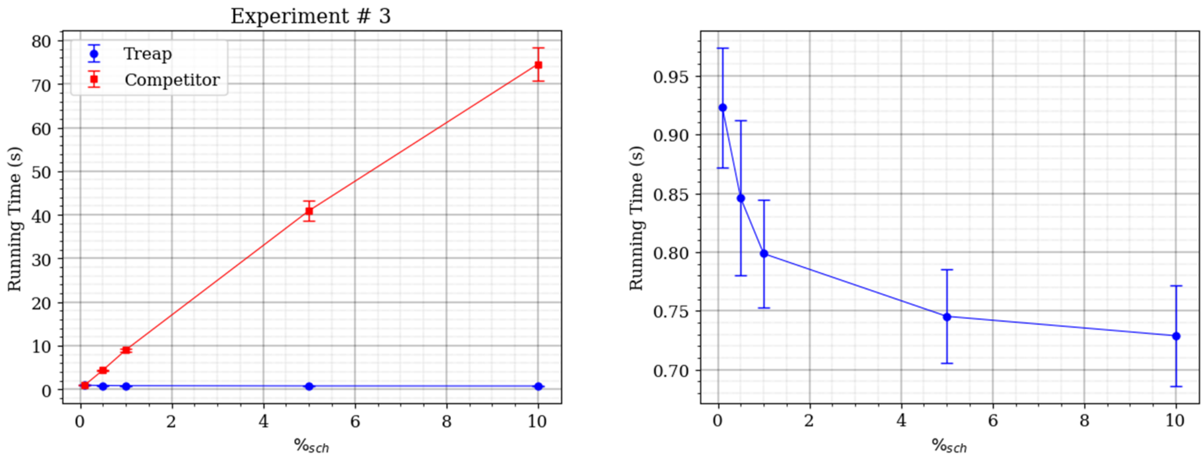
$$E[C] \quad \in \quad (1-p)L \cdot O(1) + p(1-2p) \cdot O(1)\sum_{i=1}^{L} i \approx \left[\frac{L(L-1)}{2}p + L\right] \cdot O(1) \tag{11}$$

where we've dropped a term $-L(L+1)p^2$ in the last step because p is small. Since L is a constant, we have an $O(p)$ estimated theoretical upper-bound on the total expected cost and therefore the cost is expected to increases linearly as $\%_{del}$ increases. Indeed, this behavior is what we observe in our experimental results in Figure 2(a) which shows a nearly-linear red curve corresponding to the competitor.

5.3 Experiment 3

Our third experiment is very similar to the second experiment. In this case, we allow the operation sequences to only contain insertions and search operations while keeping the sequence length fixed at $L = 1M$. We generate 5 different sequences with varying in-expectation percentages $\%sch$ of searches, choosing the values $\%sch = 0.1\%, 0.5\%, 1\%, 5\%, 10\%$. These sequences are generated by calling the $genOpSequence(L, \%del, \%sch)$ interface provided by our data generator with the arguments specified appropriately, i.e. we set $L = 1M$, $\%_{del} = 0$ and $\%_{sch}$ to one of the 5 corresponding values. Each sequence is then executed separately on both the treap and competitor. Like in our previous experiment, we perform multiple trials and report the average measured total running times in Figure 3. Figure 3(a) shows both curves for the treap and competitor. Again, as in the second experiment, we see a large discrepancy between the treap and competitor running times, the running times for the treap are relatively negligible and difficult to see in the plot. So we have also shown a a close-up view of the treap curve only in Figure 3(b).

From Figure 3(a), we can see that the measured running time of the competitor grown nearly linearly with $\%sch$. While from Figure 3(b), there is clear indication that the running time of the treap decreases as $\%sch$ increases. Intuitively this makes perfect sense. In the case of the dynamic array, each search operation cost $O(n)$ in the worst-case while each insertion costs $O(1)$ amortized, i.e. searches are far more expensive than insertions. As $\%sch$ increases, the expected number of search



(a) (b)

Figure 3: Time vs. Search Percentage: (a) Treap and Competitor (b) Treap only

operations in the sequence increases and therefore the expected total cost of the sequence for the dynamic array should increase as $\%sch$ increases.

On the other hand, it is also true that as the expected number of searches in the sequence increases, we expect fewer insertions to occur, so the expected number of elements in the data structure should decrease as $\%sch$ increases. Since the expected cost of each operation on a treap is $O(\log n_i)$, this means operations are cheaper in expectation when the treap contains fewer elements. Therefore, we would expect that as $\%sch$ increases, the treap size will be smaller in expectation, so the expected total cost of the operation sequence should decrease. Moreover, we also note that just like in our previous experiment, the treap is roughly 100x faster than the competitor when $\%sch = 10$.

We can do some mathematical analysis to make our intuitions more concrete. Let's start by considering the dynamic array. Let $L = L_{ins} + L_{sch}$, where L is the length of the operation sequence, L_{ins} is the total number of insertions and L_{sch} is the total number of searches. Then the total worst-case cost of the operation sequence can be decomposed into a sum of two terms. The first term is the sum over the cost of all insertions and the second term is a sum of costs over all searches.

$$\begin{aligned} C &= \sum_{i=1}^{L} cost(o_i) \in \sum_{\{o_i\} \wedge o_i = insertion} O(1) + \sum_{\{o_i\} \wedge o_i = search} O(n_i) \\ &\leq \sum_{i=1}^{L_{ins}} O(1) + \sum_{i=L_{ins}+1}^{L} O(n_i) \end{aligned} \tag{12}$$

where n_i is the number of elements in the data structure during the ith operation, as usual. The less than or equal to sign on the second line is obtained by observing that this upper bound corresponds to the largest possible total cost which is incurred only when the first L_{ins} operations in the sequence are all inserts and the last L_{sch} operations in the sequence are all searches[5]. In that case, because

[5] After all insertions have been completed, the array will have reached it's largest possible size and then performing the searches will incur the largest possible cost because a single search has $O(n_i)$ worst-case cost.

no more insertions occur after the first L_{ins} operations, the number of elements in the array should remain constant throughout the rest of the sequence. In other words, we have the following relation:

$$n_i = \begin{cases} i & \text{if } i \leq L_{ins} \\ L_{ins} & \text{if } i \geq L_{ins}+1 \end{cases} \tag{13}$$

Substituting this relation into Equation (9), we obtain:

$$\begin{aligned} C &\in \sum_{i=1}^{L_{ins}} O(1) + \sum_{i=L_{ins}+1}^{L} O(1) \cdot L_{ins} \\ &= (L_{ins} + L_{ins} \cdot L_{sch}) \cdot O(1) \\ &= (L - L_{sch})(1 + L_{sch}) \cdot O(1) \end{aligned}$$

Since L is large (i.e. $L >> 1$), we can use the approximation: $L_{sch} \approx pL$, which is just the number of search operations in the sequence in expectation. Then we have the following:

$$C \in (L - pL)(1 + pL) \cdot O(1) \approx [L(L-1)p + L]\, O(1) \tag{14}$$

In the last step, we have used the approximation $L^2p - L^2p^2 \approx L^2p$ which is valid since p is small. Since L is a constant, then according to this analysis, we have an $O(p)$ estimated theoretical upper-bound on the worst-case total cost and therefore the cost is expected to also increase linearly with $\%sch$ because $p = \frac{\%sch}{100}$. This supports the experimentally observed nearly-linear curve for the competitor, as shown in Figure 3(a).

Now lets consider the treap. Given L_{ins} is the total number of insertions and L_{sch} is the total number of searches, we can observe that since the the treap is initially empty and number of elements in the treap at the end of the sequence will be exactly equal to L_{ins}, then the expected height of the treap must be upper-bounded by $O(\log L_{ins})$. Because all possible permutations of a sequence containing L_{ins} insertions and L_{sch} searches should lead to this same upper-bound on the expected treap height, we can arbitrarily choose any permututaion for our analysis. In particular, we can pick the sequence in which all insertions occur first followed by all searches, which will simplify our analysis because we can use the relation from Equation (10). The the expected total cost of the sequence can be expressed as follows:

$$\begin{aligned} E[C] &\in \sum_{i=1}^{L} O(\log n_i) = \sum_{i=1}^{L_{ins}} O(\log i) + \log(L_{ins}) \sum_{i=L_{ins}+1}^{L} O(1) \\ &\approx L_{ins}\log(L_{ins}) + L_{sch}\log(L_{ins}) = L\log(L_{ins}) \end{aligned} \tag{15}$$

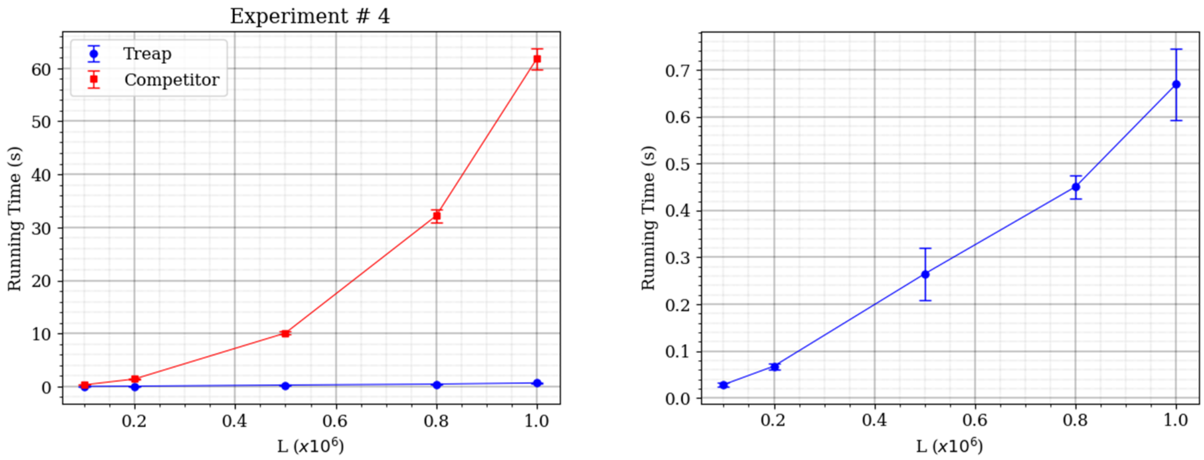
where we have used Stirling's approximation, i.e. $\sum_{i=1}^{L_{ins}} O(\log i) \in L_{ins}\log(L_{ins})$ for large L_{ins}. According to this equation, we have $E[C] \in O(\log L_{ins})$. Thus the total expected cost of the operation sequence depends logarithmically on L_{ins} so that a decrease in L_{ins} would cause a corresponding decrease in the cost. For large L, we expect $L_{ins} \approx (1-p)L$ which then implies the following:

$$E[C] \in L\log(L_{ins}) \approx O(L\log(L) + L\log(1-p)) \tag{16}$$

Therefore we have an $O(\log(1-p))$ estimated theoretical upper-bound on the expected total cost, which is a decreasing function of p. So an increase in $\%sch$ which corresponds to an increase in p will cause the expected total cost of sequence to decrease, confirming our intuition and also the observed measured running times for the treap shown in Figure 3(b).

5.4 Experiment 4

In our final experiment, we generate 5 different sequences with varying sequence lengths chosen as follows: $0.1M$, $0.2M$, $0.5M$, $0.8M$ and $1M$. Each sequence contains insertions, deletions and searches with in-expectation percentages fixed at $90\%, 5\%$ and 5% respectively. The sequences are generated by calling the $genOpSequence(L, \%del, \%sch)$ interface provided by our data generator with the arguments specified appropriately, i.e. we set $\%_{del} = 5$, $\%_{sch} = 5$ and L to one of the 5 corresponding values. Each sequence is then executed separately on both the treap and competitor. Like in our previous experiment, we perform multiple trials and report the average measured total running times in Figure 4. Figure 4(a) shows both curves for the treap and competitor. Again, as in previous experiments, we see that the running times for the treap are relatively negligible and difficult to see in the plot. So we have included a close-up view of the treap curve only in Figure 4(b).



(a) (b)

Figure 4: Time vs. Length of Mixed Operation Sequence: (a) Treap and Competitor (b) Treap only

Figure 4 indicates that the running times for both data structures increases as sequence length increases. Once again, the treap is about 100x faster when $L = 1M$ and vastly outperforms the competitor. These results are not surprising. Given that deletion and search operations are very expensive for the dynamic array, being $O(n)$ worst-case, having even a relatively small percentage of either or both of these operations in a sequence can quickly dominate the total running time. This is a common theme that we've been observing over the last three experiments, i.e. Experiments 2,3 and 4. For the treap, the cost of all three operations are roughly the same $O(\log n)$ and the total running time can be predicted almost entirely based on the number of elements contained in the treap at the end of an operation sequence. For sequences with fixed in-expectation percentages for the three operations, as sequence length increases, we would expect the number of elements in the treap to also be larger at the end of the sequence. This is why in both Experiments 1 and 4, we observe the total running time of the treap to increase with sequence length. Whereas in Experiments 2 and

3, we observe the opposite effect. As discussed in previous sections, this happens precisely because of the following reason: if the relative percentages of the delete and/or search operations increases while relative percentage of insertions decreases, the expected number of elements in the treap will be smaller at the end of the operation sequence which corresponds to decreasing total running time.

We would now like to show for this experiment, using some mathematical analysis, that the asymptotic growth of the dynamic array is much faster than that of the treap as sequence length becomes larger. Instead of starting from scratch, we can borrow from much of the analysis that was carried out for experiment 2, making slight modifications. Let's define a random variable X_k which equals 1 if the kth operation is an insertion, -1 if the kth operation is a deletion and 0 if the kth operation is a search. Then we can define the probabilities $Pr[X_k = -1] = Pr[o_k = deletion] = p$, $Pr[X_k = 0] = Pr[o_k = search] = p$ and $Pr[X_k = 1] = Pr[o_k = insertion] = 1 - 2p$. Then we can express the total number of elements in the data structure after the ith operation as follows:

$$n_i = \sum_{k=1}^{i} X_k$$

Then the expected total number of elements following the ith operation is given by:

$$E[n_i] = \sum_{k=1}^{i} E[X_k] = \sum_{k=1}^{i} Pr[X_k = 1] - Pr[X_k = -1] = \sum_{k=1}^{i}((1-2p)-p) = (1-3p)i \tag{17}$$

When i is large, we can invoke the approximation $n_i \approx E[n_i] = (1-3p)i$. Substituting this into Equation (7) gives us the following expected total cost for the treap:

$$E[C] \in O\left(\sum_{i=1}^{L}(\log(1-3p) + log(i))\right) \approx O\Big(L\log(1-3p) + L\log(L)\Big) \tag{18}$$

Next, we consider the worst-case cost of the ith operation for the dynamic array, which is given by the following:

$$c_i = cost(o_i) = \begin{cases} O(1) & \text{if } o_i = insertion \\ O(n_i) & \text{if } o_i = deletion \vee search \end{cases} \tag{19}$$

where n_i is the number of elements in the dynamic array during the operation. Then the total expected cost for the entire sequence of operations is given by:

$$\begin{aligned} E[C] &= (1-2p)\sum_{i=1}^{L} cost(insertion) + p\sum_{i=1}^{L} cost(deletion) + p\sum_{i=1}^{L} cost(search) \\ &\in (1-2p)\sum_{i=1}^{L} O(1) + 2p\sum_{i=1}^{L} O(n_i) \end{aligned}$$

Invoking the approximation $n_i \approx E[n_i] = (1-3p)i$, we get the following:

$$\begin{aligned} E[C] &\in (1-2p)L \cdot O(1) + 2p(1-3p) \cdot O(1)\sum_{i=1}^{L} i \\ &= \left[L^2(1-3p)p + L(1-p-2p^2)\right] \cdot O(1) \end{aligned} \tag{20}$$

Given that $p = 0.05$ is a fixed constant, we have an estimated theoretical upper-bound of $O(L \log L)$ for the total running time of the treap and $O(L^2)$ for the dynamic array. Thus the asymptotic growth of the total running time of the dynamic array is expected to be much faster than that of the treap. Indeed, this is observed in our experimental results. Figure 4(a) even indicates what appears to be a quadratically growing curve for the dynamic array and Figure 4(b) indicates a slightly faster than linearly growing curve for the treap.

A lot of our mathematical analysis so far made use of approximations and we were primarily concerned with the asymptotic behavior of expected total costs. We have provided valid justifications for the use of those approximations and it is also useful to realize that the law of large numbers[6] guarantees, to some extent, that total cost in expectation will be very close to the true total cost. In our case, the true total cost can be modeled as a sum over a large number of independent and almost-identically-distributed random variables.

6 Conclusion

Our experimental study has revealed that both the treap and competitor data structures perform in close accordance to theoretical expectations, even under a non-idealized practical setting. The experiments have been set up to resemble ”realistic” application scenarios under which these data structures are expected to be employed. Such as scenarios in which large sequences of operations may be performed and the sequence may contain different relative proportions of the different types of operations. Our first experiment has revealed that in the case where the operation sequence contains only insertions and no deletions or searches, the competitor is able to easily outperform the treap. This result is obvious in light of the fact that the insertion operation is significantly cheaper for the competitor, i.e. $O(1)$ amortized compared to $O(\log n)$ in expectation for the treap. Thus, under a hypothetical application scenario in which we only ever need to perform insertions, a dynamic array would be a better choice over the treap.

All three of our remaining experiments have revealed that given a sequence containing mostly insertions, introducing even a small proportion of deletions and/or searches into the mix, e.g. less than 1%, can drastically turn the tables in favor of the treap. This is a direct consequence of the fact that the worst-case cost of the deletion and search operations on a dynamic array is $O(n)$ which is exponentially larger than the corresponding $O(\log n)$ expected cost for these operations on a treap. To make this statement more concrete, we have also carried out mathematical analysis in Section 5 to estimate the asymptotic behavior of the total running-times with respect to the parameters which where varied during each experiment and showed faster asymptotic growth of the total running time for the dynamic array in comparison to the treap. This leads us to conclude that for application scenarios in which large sequences of operations need to be performed on the data structure and these operations may contain a mix of insertions in addition to deletions and/or searches, then the randomized treap will be the better choice as it is expected to have superior performance over a dynamic array.

It is also worth noting that much of our theoretical analysis relied on the assumption that the

[6] The large number in this case being the length of the operation sequences.

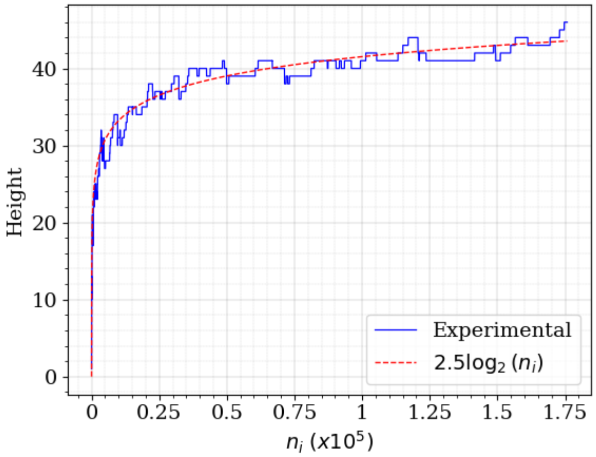


Figure 5: Treap Height vs. Treap Size. Blue curve shows the actual height of the treap and dashed red curve corresponds to $2.5\log_2(n_i) \in O(\log(n_i))$

actual performance of the treap is very close to its expected performance, in other words, the height of the treap remains roughly $O(\log n)$ over the course of any large operation sequence. We have already shown definitively that the experimental results closely matched with our theoretical analysis. This serves as evidence for the actual performance of the treap indeed being very close to the expected performance. To obtain further concrete evidence, we have tracked the evolution of the height and size of the treap during the σ_2 sequence from Experiment 4 and present this data in Figure 5. Indeed, it is clear from the plot that the actual height always stays within a constant factor of $\log(n)$, in particular the actual height $\approx 2.5\log_2(n_i)$, where n_i is the treap size during the *ith* operation in the sequence. In fact, the probability that the height of a randomized treap exceeds $O(\log n)$ scales as $\frac{1}{n^f}$ where f is a positive constant, e.g. see [1] for proof. In other words, for large n, the height of a treap tends to be $O(\log n)$ with very high probability and so the true cost per-operation on a treap tends to be very close to $O(\log n)$, which confirms our experimental observations.

References

[1] Sariel Har-Peled, *Class Notes on Randomized Algorithms*, 2024, `https://sarielhp.org/teach/notes/algos/files/book.pdf`.