Project Report

Treap Data Structure - Implementation and Verification in Dafny

Students:

Achmad Yogi Prakoso - A0250692A

Ng Geon Woo Robin - A0218422M

CS5232 - Formal Verification and Design Techniques

School of Computing | National University of Singapore (NUS)

2023

Contents

1 Introduction

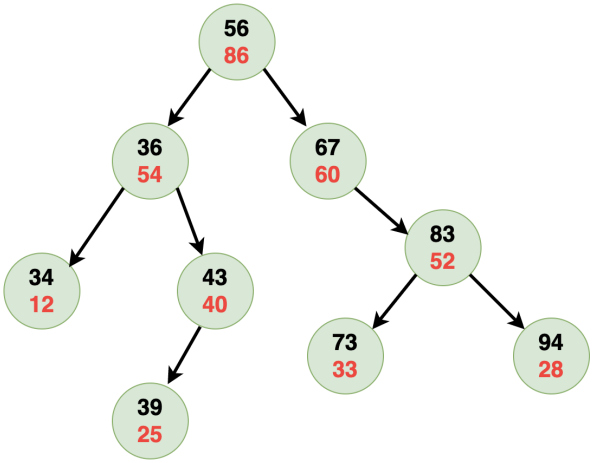

Figure 1: This figure gives an example of treap data structure. The black number is the key, whereas the red number is the heap priority

A treap, also known as a Cartesian tree, is a data structure that combines the features of a binary search tree (BST) and a binary heap. The combination allows the tree to have an expected logarithmic depth of its number of nodes so that, with high probability, the tree is balanced. It also performs less than two rotations for updating its contents [3], which makes it more efficient compared with the deterministic counterparts. Its randomness property makes it efficient to maintain a dynamic set of ordered elements while allowing for fast search, insertion, and deletion operations. During these operations, treap does not guarantee its topology to be consistently balanced. Hence, one must be careful when using this data structure if a specific time-bound is required. Treaps are particularly useful in cases where the data set changes over time and we want to maintain the elements in a sorted manner.

In a treap, each node has two attributes: a key and a priority. The key is used to maintain the binary search tree property, that is to ensure for each node, all the keys in its left subtree are smaller and all the keys in its right subtree are larger. As the keys hold an essential role in searching the data, we keep the keys unique. The priority, on the other hand, is used to maintain the heap property which controls the balance. For this purpose, we may allow duplicates up to some limits. For maintaining the property, we can use max-heap or min-heap property. In this project, we used max-heap property as demonstrated in Figure 1 – The priority of a node is at least the same as its children's priority.

The expected height of the treap is $O(\log n)$, which means that the search, insertion, and deletion operations are efficient in practice. However, there is no guarantee that a treap will always achieve logarithmic height because of the randomized priority. The worst-case complexity of treap operations can be $O(n)$, which is idential with a linked-list. This bad scenario occurs when the priorities lead to a highly unbalanced tree. Despite having some drawbacks, treaps are often used because of their simplicity and good average-case performance.

1.1 Treap Operations

Treap operations are just the same as operations used in binary search tree. The basic operations includes build, insert, delete, search, split, and merge. There are also other operations such as union and intersect. But we will only discuss the six previously mentioned operations as this project can handle so far.

Insert - The insert method takes a key as an input. The key can be in various forms of data types. For simplicity, let's take an integer as the input. The heap priority, which is also an integer, will be created from the input in a deterministic manner. We can deploy a hash function to perform this operation. Once the method populates a key and a priority, it can start traversing the node to find an appropriate position. A rotation might perform to restore heap property. The average complexity of this operation is $O(\log n)$, but it can be as bad as $O(n)$ if the tree forms a linked-list. Therefore, picking a good hash function for heap property is essential to avoid forming an unbalanced tree.

Build - The build method takes an array of keys as an input. It basically performs insertions for multiple keys in a row. Therefore, we can simply call insert method for the build implementation. Depending on how the tree is structured, it can take a linear time $O(n)$ if the input array is already sorted. However, sorting an array can take $O(n \log n)$ times assuming we are using merge sort. In this project, we will not limit the input array such that it can be in a random order and has duplicates. Hence, the overall time complexity would be $O(m \log n)$ where m is the array length and n is the total available nodes.

Delete - The delete method takes a key as an input to find the corresponding node. We can only delete a node if it is a leaf or one of its children is *null*. Therefore, some rotations perform to bring the node down to the leaf which can take an average time of $O(\log n)$.

Search - Using an input key, the search operation is quite straightforward as we already use it during the insert and the delete operation. It takes an average of $O(\log n)$ times to find the node.

Split - The split operation divides a treap into two treaps using an input key such that the left treap keys are at most the input key, and the right treap keys are greater than the input key. To implement split, we first search it. Either found or not found, we can simply disconnect the relation of the parent node and its right child. Since the right child's left child has lesser key (if not *null*), we can promote it to become the right child of the current parent node. The averate time complexity to perform this operation is $O(\log n)$.

Merge - The merge operation takes two treaps to form one treap. The operation is actually reverting two trees back to one after splitted. Having said that, it is assumed that the input treaps are

well-ordered such that all keys in the left treap are smaller than all keys in the right treap. If two treaps do not have the ordered property, we can use *union* operation to put them together into a single treap, but we will not discuss *union* in this report. To perform merge operation, we split the right treap using the root's key from the left treap to get a new left and right treap. We then merge them with the corresponding left and right child of the root of the left treap. We keep doing these activities recursively until it gets into leaves. The averate time complexity to perform this operation is $O(\log n)$.

1.2 Treap Applications

Treaps have been used in various scenarios due to their simplicity and strong average-case performance. Among the frequent use cases for treaps are dynamic ordered sets, range queries, priority queue, and many more. Treaps has the capacity to keep a dynamic set of ordered elements, making search, insertion, and deletion operations effective. This qualifies them for uses like keeping a sorted list of users, things, or events, which involve frequent modifications to the data collection.

There are some research regarding the application of treaps. Blelloch and Margaret proposed treaps to build fast set operations such as union, intersection, and difference under $O(m \log(n/m))$ time complexity [2]. For memory appliction, since memory nowadays is getting large, treap is used in memory indexing to better interact with caches. Although memory performance is considered fast, it is still slow compared with on-chip caches. Another interesting application was also suggested by Dharya and Shalini, who proposed using treaps instead of adjacency matrix to optimize graph storage. [1]. Based on those applications, we decided to formalize treaps as a part of this project. In the following sections, we will go through the details on how we formalized and verified some operations in treaps.

2 Project Results

In this section we will explain about Treap Data Structure specific to our implementation. To get started, we have posted our project into Github with public access so that anyone can visit. We released our first and, probably, also the last version named *v1.0* that you can check here. Our treap data structure is organized into two Dafny files called *treapNode.dfy* and *treap.dfy*. You can check the *readme* file to explore further. In general, we applied the following design as shown in Figure 1.

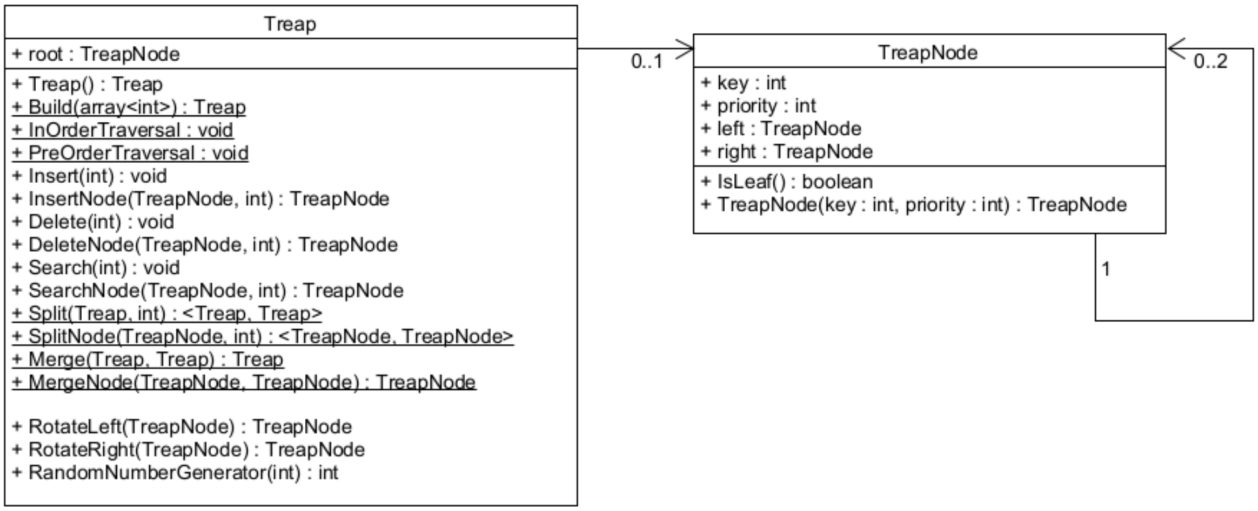

Figure 2: Treap and TreapNode Class

The diagram shown in Figure 1 is the UML class diagram of the Treap and TreapNode. The TreapNode encapsulates the node of a Treap, storing integer values for the node's associated key and priority, as well as pointers to the left and right child TreapNodes. The Treap class holds a pointer to the root TreapNode and defines the methods associated with a Treap.

Some operations are static methods (those with underlines) as the nature of their functionality that do not affect a treap object instance. For example, insert method is a non-static method because it is intended to update a treap instance's contents. However, split implementation uses static method as it is intended as a tool to create two new treaps from one treap input. It will not be effective if we declare split as a non-static method because we need to create a new treap instance before we can use the tool. It also has nothing to do with the newly created instance.

2.1 Ghost Variables and Predicates

Apart from the concrete fields and methods as shown in the diagram, the classes also define several ghost variables and predicates that are only used to perform verification and will not be part of the compiled program. This section will list and explain the rationale of these ghost fields.

Class *TreapNode* contains two *Ghost* variables called *Repr* and *Values*. *Repr* has been a gen-

eral practice in Dafny to provide a decrease counter in a recursive call so that Dafny understands where the program should terminate. It is a set of objects, which are instances of *TreapNode*s of the node itself and its all descendants. Variable *Values* is another *Ghost* variable that stores all keys of the node and its all descendants. The later variable is important to validate the binary search property.

```
class TreapNode {
    var Repr: set<TreapNode>;
    var Values: set<int>;

    ghost predicate Valid() {}
    ghost predicate ValidHeap() {}
    ...
}
```

A critical message for the node design is that each node can only access its children. It has no access to know what its parent is. This situation restricts our ability to validate a binary search property because we cannot say that the right child's key is strictly less than the parent's. Variable *Values* solves this issue as the keys of a node's descendants are stored in it, improving its parent's visibility to check that all keys in its left descendants are strictly less than its value, and vice versa.

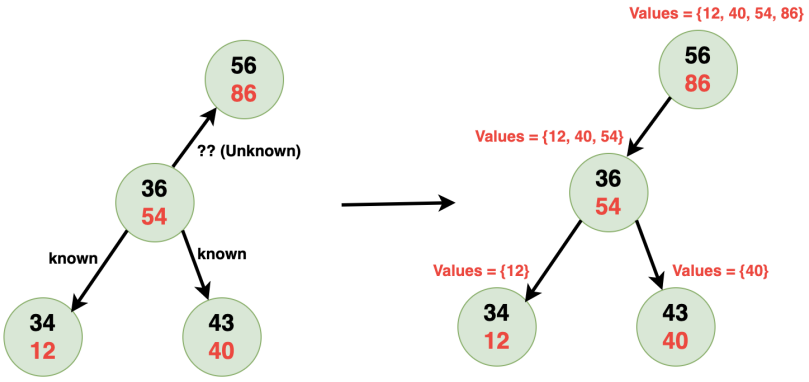

Figure 3: An illustration on how variable *Values* helps a node to verify binary search property in a recursive call

Class *TreapNode* has two *Ghost* predicates: *Valid* and *ValidHeap*. Predicate *Valid* is used to verify the binary search property as well as giving Dafny a helper for recursive calls. Whereas predicate *ValidHeap* is used to verify the heap properties. It might seem weird to know that our design requires two verification predicates. In fact, the separation is necessarry to provide a compensation in some certain scenarios to have a state violation. In any change of states, binary search properties must always hold, but it is not mandatory for the heap properties – The design will eventually force the later scenario to be valid. One simple case is during the rotation. Rotation requires the input node's heap to be invalid, or else we have no reason to rotate the nodes. An invalid heap can happen after insertion where the parent of the new node realizes that it has lower heap property.

Now we are moving to the next class, *Treap*. Class *Treap* plays a role as a real tree that gives a shelter to nodes by knowing their root node. In this class, all operations take place. The *Ghost* variables and predicates are just the same as *TreapNode*, except that it has no predicate *ValidHeap*. *Repr* and *Values* have the same contents as *TreapNode* just that $|TreapNode.Repr| == |Treap.Repr| + 1$ with additional instance from the tree itself. The copies are imperative to verify that the tree is holding the correct root node. Furthermore, class *Treap* requires all properties to be valid in all states so that it has only one *Ghost*predicate (*Valid*).

```
class Treap {
    var Repr: set<TreapNode>;
    var Values: set<int>;

    ghost predicate Valid() {}
    ...
}
```

2.2 Available Methods

The implemented methods are available in two versions: a main method that interacts with the Treap at a higher level, and a Node version of the same method that interacts with individual TreapNodes. This separation allows us to define pre- and post-conditions separately at the Treap and TreapNode levels, ensuring the validity of the Treap before and after the main methods are executed, as well as the validity of the desired properties of individual nodes during individual recursive calls to the Node-level methods.

The implementation details of methods will not be mentioned in detail in this report as it is very similar to commonly available concrete implementations of balanced binary search trees utilising rotations to maintain balance, such as AVL trees, in other programming languages like Java and C++. The only additional statements within the implementation of methods are the updating of the appropriate ghost variables to ensure consistency between the logical and concrete representation of the data structure.

We will instead draw attention to the pre- and post-conditions of our implemented methods to provide a comprehensive overview of the guarantees that each method offers and clarify the purpose of specific post-conditions, and the role they play in verifying the correctness of our implementation. Our objective is to offer a detailed explanation of the pre- and post-conditions in order to provide a better understanding of how they work together to ensure the correctness of our program. We have listed the verification details for pre- and post-conditions in Appendix A.

2.3 Property Value Generator

The only factor to balance a treap is its heap priority. Picking an "appropriate" value will secure the treap from being highly unbalanced. For example, if the priority increases as the key gets higher, it will led to a linked-list topology. Therefore, on increasing key, the priority must behave either going up or going down under roughly the same probability. The tools for generating random number can be arbitrary. In this project, we use the following formula to generate random numbers from an input key.

$$\text{priority} = (\text{val} * \frac{654321}{123})\%1000 \tag{1}$$

There is nothing special about the formula. It is just an operation of arbitrary numbers to assign at most a thousand distinct possible priorities from a given input value. We can check its distribution from Figure 4. The formula gives results that oscillate up and down systematically over the range 0 to 1000. An oscillation indicates that collision exists, and the pattern will recur after some intervals.

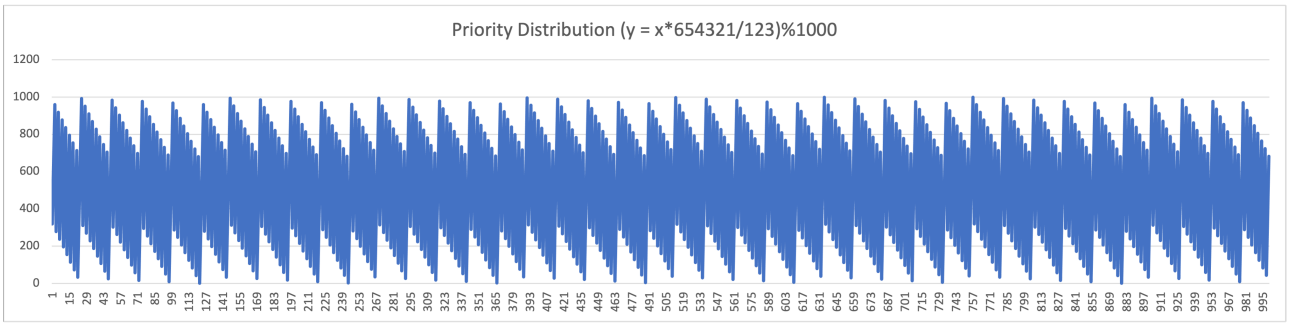

Figure 4: x is the first thousand keys, and y is the function result

The goal of having a hash function is to create a random behavior of heap priorities on increasing key values - We probably want a plot distribution like heartbeats. Our current priority implementation clearly is not that effective, but it is enough for this project demonstration. A better suggestion might be using cryptographic hash algorithms such as MD, SHA, RIPEMD, etc because it gives us a relatively perfect hash function. However, there are some overheads to do before we get an integer value from these algorithms. For example, an MD2 hash value consists of 16 characters length. We still need to convert those characters into a 32-bit integer value. There are many ways to do this conversion, such as playing around with bits by taking the first two bits of each character to form a 32-bit integer. Depending on how treaps are implemented, using cryptographic hash function might be heavy if fast performance is the main target. Therefore, having a simple function is also not a bad idea.

3 Discussions And Findings

3.1 Future Works

We have identified several areas that can be further developed in future iterations. Firstly, we could implement and verify additional treap methods such as "union". Secondly, the current implementation can only store integer values as keys. We can work to make the Treap data structure generic, allowing keys to be any comparable type, thus providing more flexibility and versatility to the user. Lastly, we can refactor the "Valid" predicate of TreapNode into separate "Valid", "ValidBst", "ValidHeap", and "ValidTree". This will help optimize the verification process by reducing unnecessary computations and making the naming scheme more sensible.

References

[1] Dharya Arora and Shalini Batra. Using treaps for optimization of graph storage. *International Journal of Computer Applications*, 975:8887, 2012.

[2] Guy E Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 16–26, 1998.

[3] Raimund Seidel and Cecilia R Aragon. Randomized search trees. *Algorithmica*, 16(4-5):464–497, 1996.

4 Appendix A - Pre and Post Conditions