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

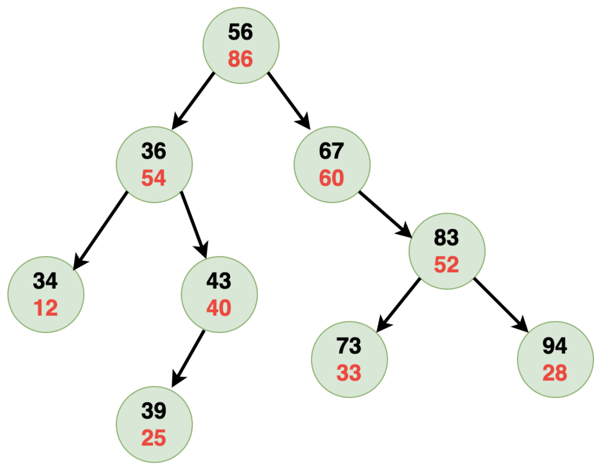

Figure 1: This figure gives an example of treap data structure. The black number is the key, whereas the red number is the heap priority

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 a logarithmic depth of its number of nodes so that, with high probability, the tree is balanced. 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.

1.1.1 Merge

1.2 Treap Applications