

Assignment 6 – Treaps

Problem Description

The purpose of the assignment was to combine the concepts of trees and heaps into the Treap data structure. It follows both the Binary Search Tree property and the Heap property. This of course means that each node has a key and a randomized priority. Any time a node is inserted, it must be rotated into a position where its priority is smaller than all of its children and its key follows the BST property. The treap program is necessary to handle insertions, deletions, and lookups in the most efficient manner. It also makes it possible to split and join two treaps.

The second part to this assignment was a test interface. Since the Treap is simply a data structure, a testing class had to be written. This part of the assignment was fairly open ended. I wrote a GUI that gave users nearly full control over the treap data structure *and* provided a number of automatic tests they could run.

The TreapMap Overview

I used systematic decomposition to break the problem into smaller parts. There are many helper functions that are reused throughout the code. The TreapNode class is a private class stored within TreapMap. It's simply the object used for each node, which contains only basic properties and functionality. Detailed descriptions of some of the more important functions are below. The only two fields that the TreapMap keeps track of are the root and the iterator node.

The Lookup Function

This function traverses recursively down the tree starting at the root. At each node, it chooses which direction based on a comparison of the current key and the key that is being searched for. It actually uses a helper function lookupNode(...) (which returns the node that is being searched for) and returns the value object.

Since it traverses straight down the tree, it has an $O(h)$ running time which, in a balanced treap, will be $O(\log n)$.

The Insert Function

After creating a new node with a random priority, this function calls the placeLeaf(...) function to insert it at a valid leaf. This function is nearly identical to the lookup function described above, finding the leaf where the new key should be placed. Because of this implementation, when it reaches a leaf, the new node will follow the BST property.

Using this method, there's a good chance that we'll have to shuffle the new node around in order to satisfy the Heap property. The insert function now calls fixPriorityUp(...), which continues to rotate the node upwards until its priority is smaller than that of its parent (or it has become the root).

The placeLeaf(...) function move straight down the tree and has an $O(h)$ or $O(\log n)$ running time. The fixPriorityUp(...) function has a worst case running time in a

balanced treap of $O(h)$ or $O(\log n)$. Therefore, the total running time for insertion is $O(h)$.

One other case to mention is the occurrence of duplicate keys. The way to avoid this problem is to simply overwrite the old value when attempting to insert the new.

The Remove Function

Removing a node is fairly simple. We first use the `lookupNode(...)` function to traverse down the tree and return the node with the given key. If the node is not found, this function returns null and removal exits. We then call `rotateDown(...)` which rotates the node downward into a leaf position. The child with the smaller priority is always rotated into the new parent during this process. To complete removal we just remove references to the node, and it is garbage collected.

The lookup function is $O(\log n)$ as described above. The `rotateDown` function, like the `fixPriorityUp` function, has a worst case of $O(h)$ or $O(\log n)$. The removal of references is $O(1)$, so the total running time is $O(\log n)$.

The Join Function

This is a destructive algorithm that joins two treapMaps together. To join them, we can then insert a useless node at a root position and set the two treaps as its left and right subtrees. If we remove the useless node, we have a valid treap that is the combination between the two treaps.

If either TreapMap is empty, it can simply consider the non-empty treap as the joined product and return with an $O(1)$ running time. Otherwise, it must make sure that the treaps are joinable. For this algorithm, one treap must contain only keys that are larger than all keys in the other.

Since it only needs to work with my implementation of treapMap, it can use `getHighestKey` and `getLowestKey` to find and compare the highest and lowest keys of each function. Using these values, we can easily determine whether the trees are joinable. By then creating and removing the useless root as described above, we can quickly complete the join.

The `get highest` and `lowest key` functions run at $O(\log n)$, as does `remove`. Therefore, this function also runs at $O(h)$.

The Split Function

This function splits the treap at a certain key into two smaller treaps. This is easy to implement because of the nature of a treap. In my implementation the smallest priority is 0. Therefore, by finding the proper key and setting its priority to -1, we can move it to the root position with existing helper functions. We can then make the left child into the root of the smaller treap, and the current root into the root of the greater subtree.

Inserting the key to split at, rotating it to the root, and removing it all run at $O(\log n)$, so the total running time of this algorithm is $O(\log n)$.

Testing Overview

I decided to put a lot of time into my testing interface for this project. I made a flexible toolkit that gave the debugger easy functions to automatically test the treap, as well as a robust set of controls that allow him to run his own tests. The only thing that

was a problem was the fact that the Treap interface didn't give any way to alter a node's priority.

All of the tests in my program used integers for keys and strings for values, because it made the output easy to follow. Most of the output is actually sent to txt files to improve running time and ease-of-use.

The basic abilities of TreapTester allowed a user to insert, remove, and lookup nodes in a treap. The program actually stored two treaps (A and B), so that the user could switch between the two to test splitting and joining. It provides simple output to file or to the console. It also allows stress testing by inserting every word in the 'words.txt' from the boggle assignment into the treap. Quick and easy insertion can also be made by inserting random key/value pairs.

The automatic tests are as follows: "Check BST" traverses through the tree and makes sure that each node follows the BST properties. "Test iteration" simply makes sure that the iterator properly traverses from the lowest to the highest key. "Test Split and Join" saves a copy of the treap, splits it, and then rejoins the parts. It then checks to make sure the resultant tree is identical to the original. Similarly, "Test insert and remove" inserts a given key/value pair, and then removes it. It checks to see that the resultant tree is as expected.

The "Run test suite" option performs a series of test cases that could commonly produce errors and reports the results. Of course these cases and all others could be tested by the user through the other tools provided. This is just a convenience for the user. These cases include:

- Null key or value inserted
- Duplicate key inserted
- Non-existent key removed
- Tree size of 1 or 0 split
- Splitting a tree along non-existent key
- Trees of size 0 or 1 joined
- Joining unjoinable trees

Results

This program was a good way to make sure I knew all about tree rotations and manipulations, as well as the concepts of heaps and BSTs. I ended up finishing the TreapMap rather quickly and spending a huge amount of time on my testing program.

When I began writing the treap, the tester was just a small set of functions (insert, remove, and print) that allowed me to see the basics. As I continued working, I evolved the test program to a console program with basic user interaction and a few more features. After I'd finished the basics of treap and the number of options in the test program grew, I converted it into a GUI and used it to test my own treap implementation. From there, I just added any extra tests that I could think of. I was very pleased with the way it turned out.

Contributions

I talked to Fei briefly about the about the basic concept of the assignment. We were sure to not talk about any code or our implementations.