CMSC 420: Treaps

Justin Wyss-Gallifent

November 8, 2023

1 Introduction

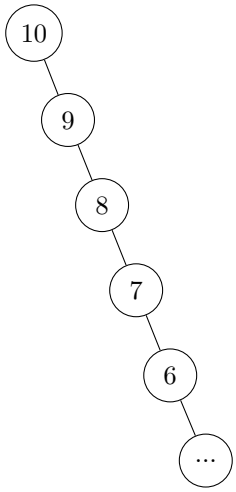
One solution to the fact that a binary search tree can become unbalanced is to rebalance it after insertion and deletion as we saw with AVL trees.

This is not the only way, and a treap is another way whose performance is anchored in a certain degree of randomness.

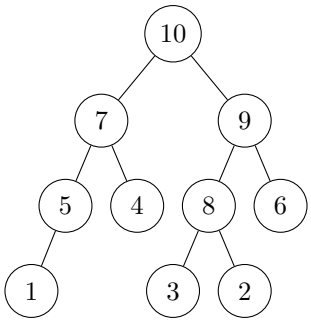
For a moment let's forget about binary search trees and recall max heaps. A max heap is a complete binary tree in which each node's value is larger than all the values in its subtree.

Let's drop the "complete" requirement and just consider binary trees in which each node's value is larger than all the values in its subtrees. We'll say that this binary tree satisfies the *max heap order*. and let's call the heap values *priorities*.

Given a list of priorities such as $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$, a binary tree which contains these priorities and satisfies the max heap order is much like a binary search tree in the sense that it could be very unbalanced such as:



Or it could be quite balanced such as:



Theorem 1.0.1. On average a binary tree which satisfies the max heap order will be fairly balanced. By this we mean the expected height will satisfy $h =$

$\Theta(\lg n)$ where n is the number of nodes/keys.

Proof. The proof of this is not trivial but should feel intuitive. $\mathcal{QED}$

Note 1.0.1. It's important to understand that we could use min heap order instead of max heap order for this. David Mount's notes use min heap order so I decided to use max heap order for some variety.

2 Treap Definition

Definition 2.0.1. A *treap* is a tree with the property that each node, besides having a key, also has a *priority*. The tree obeys the BST property according to the keys and obeys the max heap order according to the priorities.

Theorem 2.0.1. For a given set of nodes (consisting of keys/priority pairs) the corresponding treap structure is unique provided the keys and priorities are.

Proof. The proof is loosely (constr/ind)uctive. Suppose S is a set of nodes (key/prority pairs) such that the keys and priorities are unique.

By the max heap order property the root must be the node with the largest priority.

By the the BST property the left child must contain all of the nodes with smaller keys so let $S_L \subseteq S$ be those nodes and apply this same process to S_L in place of S to get the left subtree.

By the BST property the right child must contain all of the nodes with larger keys so let $S_R \subseteq S$ be those nodes and apply this same process to S_R in place of S to get the right subtree. $\mathcal{QED}$

Note 2.0.1. If the keys or priorities are not unique then the structure may not be, either, but this doesn't affect anything else much.

Let's demonstrate partway with an example:

Example 2.1. Here is a set of key-priority pairs (k, p). The priorities here were generated randomly in $[0, 100]$.

$$\{(10, 35), (20, 10), (30, 56), (40, 42), (50, 25), (60, 17), (70, 80), (80, 65), (90, 15), (95, 32)\}$$

Let's turn this into a treap!

The top node must be the one with the maximum priority, so that's $(70, 80)$. We'll use the convention that the key is the upper value and the priority is the lower value.

For the left subtree look at all the nodes with smaller keys:

$$\{(10, 35), (20, 10), (30, 56), (40, 42), (50, 25), (60, 17)\}$$

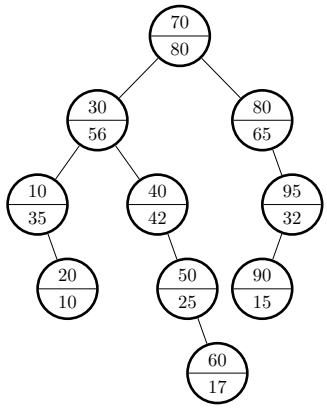
Pick the one with the largest priority, that's $(30, 56)$ so set that as the root of the left subtree. To complete that left subtree continue this process with this new list.

For the right subtree look at all the nodes with larger keys:

$$\{(80, 65), (90, 15), (95, 32)\}$$

Pick the one with the largest priority, that's $(80, 65)$ so set that as the root of the right subtree. To complete that left subtree continue this process with this new list.

The result of this is:



For a given treap then:

- When we insert we assign it a randomly chosen priority and then we restructure the tree so that it is a treap again.
- When we delete a key we also restructure the tree so that it is a treap again.

In this way we don't force the tree to be balanced but instead we give it a high probability of being balanced. It turns out that although the operations are not $\Theta(\lg n)$ in the worst case they are $\Theta(\lg n)$ in the average case.

3 Search

3.1 Algorithm

A treap is a BST according to its keys so search is precisely the same.

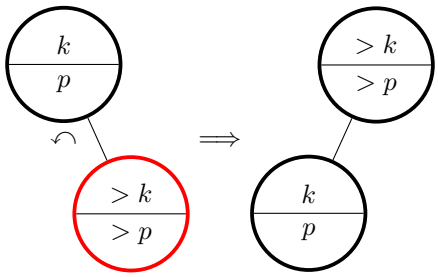
3.2 Time Complexity

The average case is $\mathcal{O}(\lg n)$ but the worse case is still $\mathcal{O}(n)$ because although on average the tree is fairly balanced there is a possibility of it being highly unbalanced.

4 Restructuring

This raises the question of how we might adjust a tree so as to correct violations of the max heap order while still preserving the BST properties. It turns out that we already know, and it's rotations!

Consider a node whose right child priority is larger than its own priority. Clearly a left rotation will fix this:



Similarly a right rotation will fix the situation of a node whose left child priority is larger than its own priority.

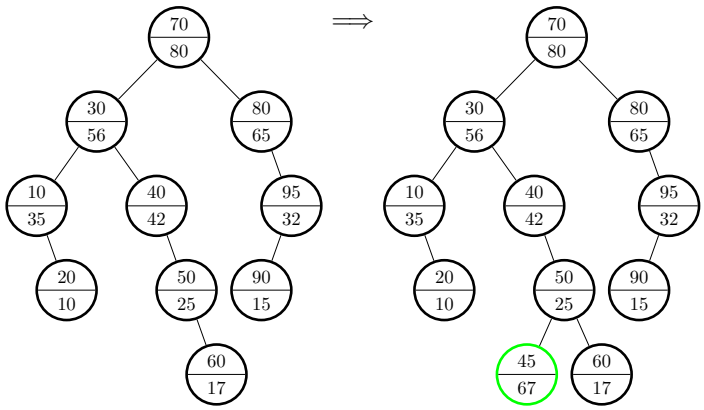
It turns out that this is all we need provided we follow the guideline of rotating so the child node with the higher priority is moved up.

5 Insertion

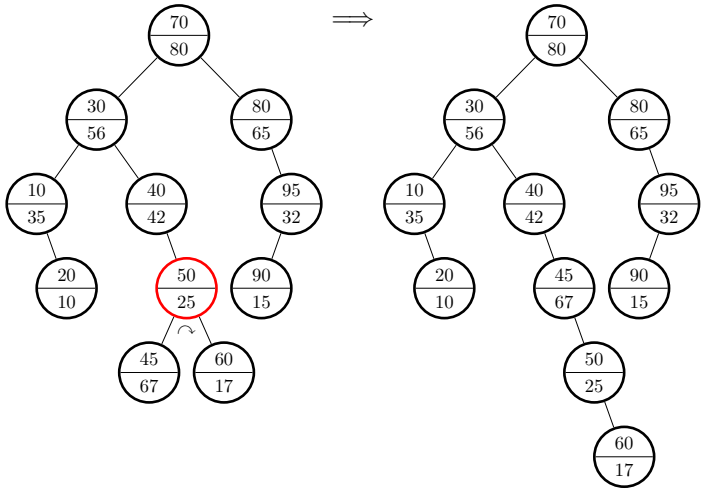
5.1 Algorithm

We insert the node with the given key as for a BST and we assign it a random priority. Then starting with the parent node we head back up the tree. If the priority of any node violates the max heap order we adjust it accordingly.

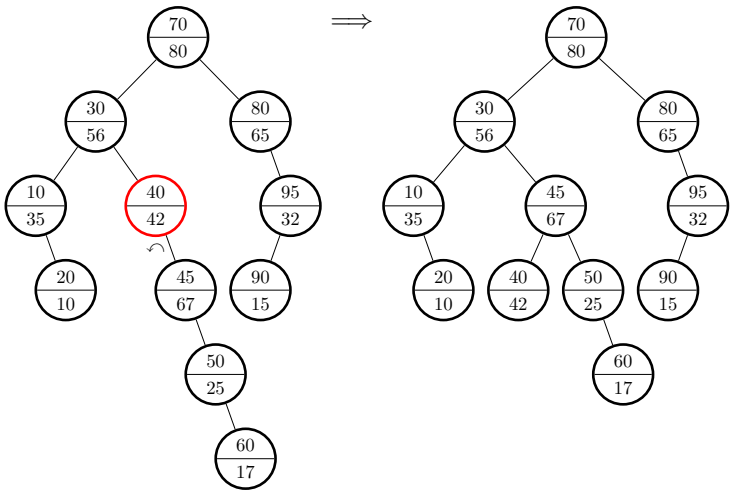
Example 5.1. Let's insert the key 45 into our earlier tree. We randomly assign it a priority of 67. First we insert as a BST.



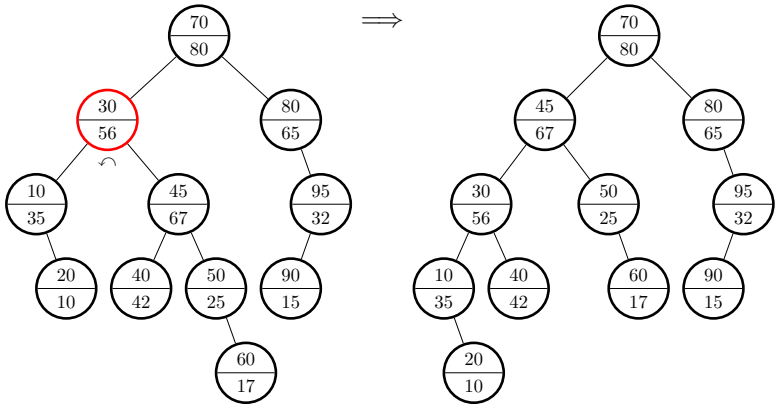
If we travel up through the tree from this node we encounter the node $(50, 25)$ which violates the max heap order. We rotate right in order to bring the higher priority 67 up:



We keep traveling up the tree and we encounter the node $(40, 42)$ which violates the max heap order. We rotate left in order to bring the higher priority 67 up:



We keep traveling up the tree and we encounter the node $(30, 56)$ which violates the max heap order. We rotate left in order to bring the higher priority 67 up:



Now we are done.

5.2 Time Complexity

Each rotation is $\Theta(1)$. For insertion, as with search, he average case is $\mathcal{O}(\lg n)$ but the worse case is still $\mathcal{O}(n)$ because although on average the tree is fairly balanced there is a possibility of it being highly unbalanced.

6 Deletion

6.1 Choices

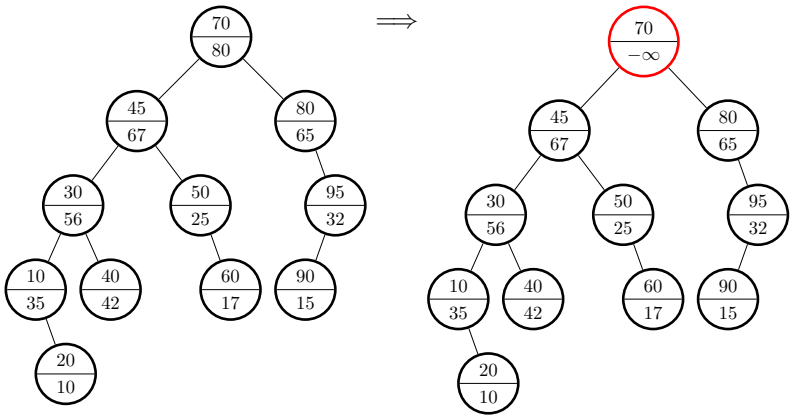
Deletion can be done a variety of ways including adapting the standard replacement approach with some restructuring at the end. For some variety though,

here is a sneaky approach.

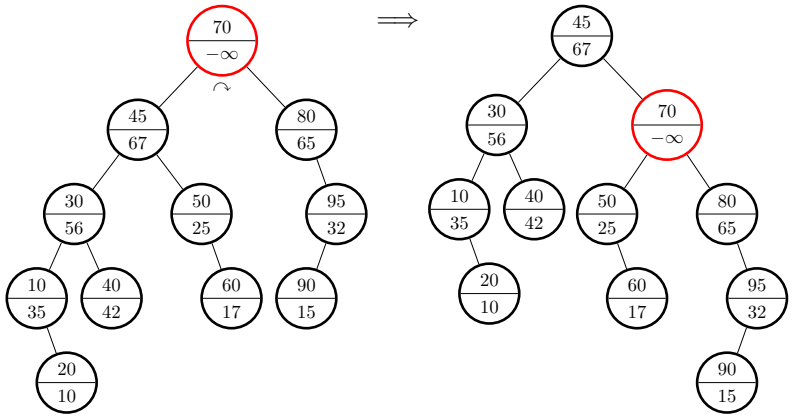
6.2 Algorithm

What we do is assign that node a priority of $-\infty$ and then use our rotations to rotate it down to a leaf where we simply chop it off.

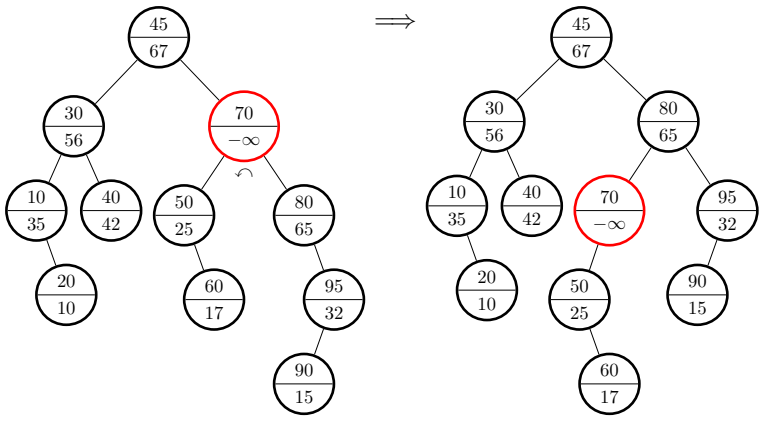
Example 6.1. Starting with the treap on the left we delete the root node by changing its priority to $-\infty$:



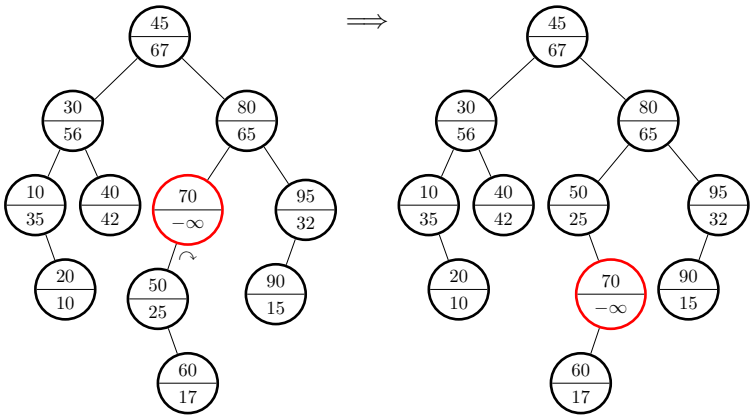
We start moving it to the bottom. First we rotate right in order to bring the higher priority 67 up:



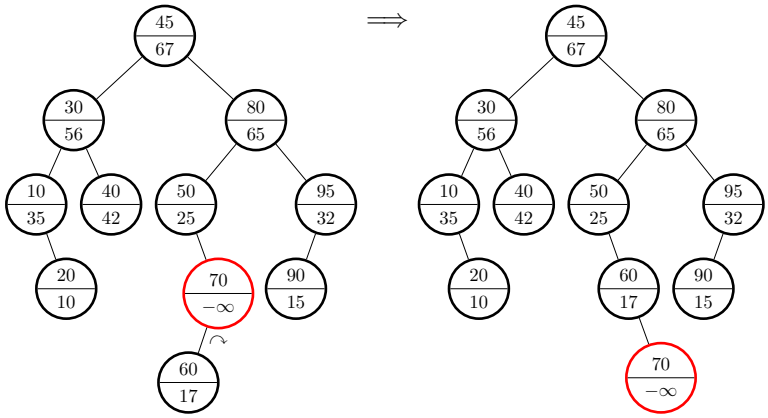
Then we rotate left in order to bring the higher priority 65 up:



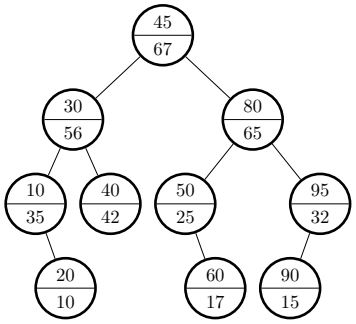
Then we rotate right in order to bring the higher priority 25 up:



Then we rotate right in order to bring the higher priority 17 up:



Now we simply chop off the node:



6.3 Time Complexity

Each rotation is $\Theta(1)$. For deletion, as with search, he average case is $\mathcal{O}(\lg n)$ but the worse case is still $\mathcal{O}(n)$ because although on average the tree is fairly balanced there is a possibility of it being highly unbalanced.