感谢互联网作者：Igor Carpanese

The treap data structure

Formally, a treap (tree + heap) is a binary tree whose nodes contain two values, a *key* and a *priority*, such that the key keeps the BST property and the priority is a random value that keeps the heap property (it doesn’t matter if it’s a max-heap or min-heap). See figure 1.

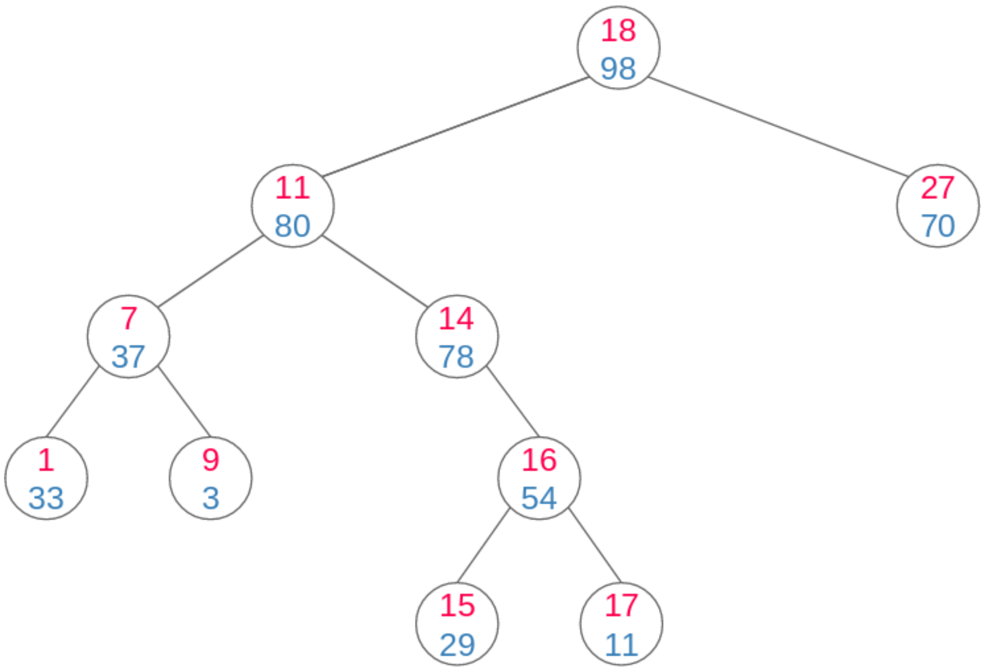

Figure 1: A example of a treap. The rose numbers are the keys (BST values) and the blue numbers are priorities given randomly (heap values).

Random Binary Search Trees

Let’s talk about one of the most beautiful interpretations of treaps.

Suppose we have, a priori, all the keys that will be inserted in a BST (a simple one, not a treap). We know that if the keys are sorted, our BST will degenerate into a list. What should we do to minimize the height of the BST if we could change the order of the insertions?

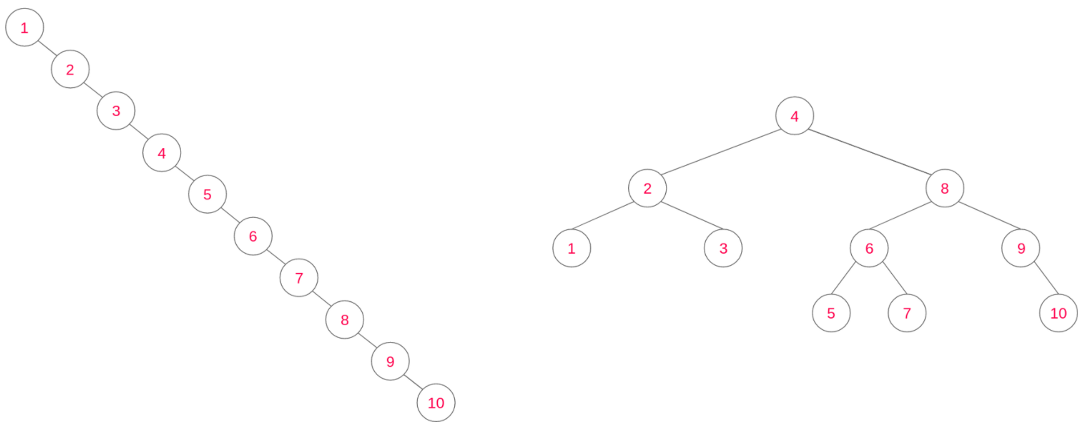

Figure 2a: It's easy to see that the order of insertions in the BST will affect the height of the tree. In the left figure (worst case scenario), we inserted elements from 1 to 10 sequentially. In the right figure (best case scenario) we inserted the elements 4, 8, 2, 3, 6, 9, 5, 7, 1, 10 in that order.

In fact, it can be prove that a simple shuffle in the list of keys is sufficient to made the tree balanced . The intuition is that the more sorted the input, the higher will be tree. The shuffle breaks the order, making the height of the tree almost minimum. In practice, we can assume that the h = lg(n).

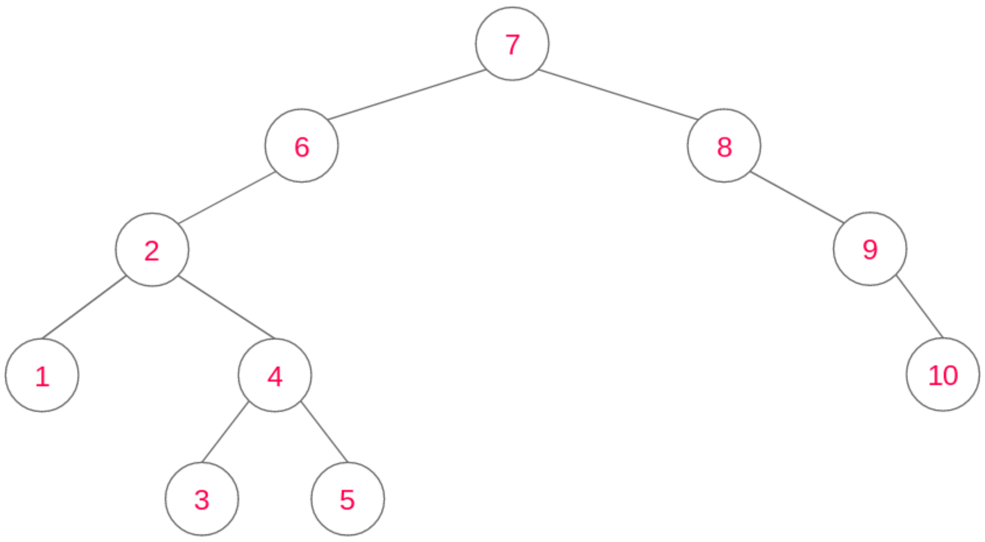

Figure 2b: After a random shuffle, the new input list was 7, 6, 8, 2, 1, 9, 4, 10, 5, 3. Note that that's not the smaller possible height, but it's way better than the worst case scenario.

That's an interesting fact, but it's also almost useless because we have to read all insertions first. The treap, however, is a data structure that can help us shuffle the keys in a smart way after each insertion.

The idea is that we can insert all the keys in the worst possible way (which means in a sorted way), but the random priority will make sure to shuffle it “in real time”.

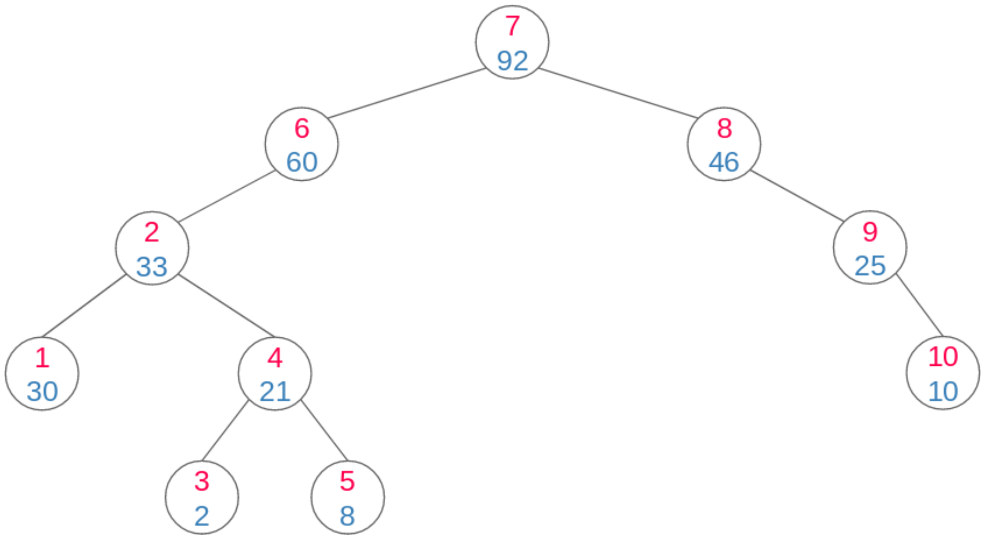

Figure 3: A treap with keys from 1 to 10. Try to figure out which key was inserted first and which key was inserted last. Was 8 inserted before 9? Do you have any clue about the order of the insertions? If you ignore the priorities, will you be able to say which key was inserted first?

We don’t know the order of the insertions in that treap (we don’t even know how to insert elements in a treap yet), but we can say for sure that the treap simulates one specific order.

Take a look at the priorities of each node. Let’s sort the vertices based on their priorities.

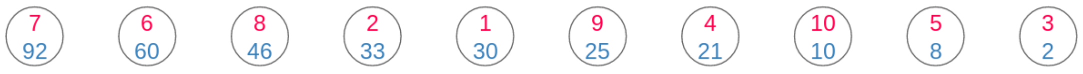

Figure 4: The nodes of the treap sorted based on their priority values (in blue).

The insertion order a treap simulates is given by the priority of its nodes. Note that, in our case, we don’t know the “real” insertion order, but the treap made a real time shuffle to 7, 6, 8, 2, 1, 9, 4, 10, 5 and 3.

In fact, the only difference between figures 2b and 3 is that the first one we shuffle a already known input, and the last one we shuffle the input in real time using random priorities.

Insertion

We already know that there could be more than one BST for a given set of nodes (see figure 2a). In fact, there's two simple operations that allows us to modify the tree and keep the BST property, the right and left rotations.

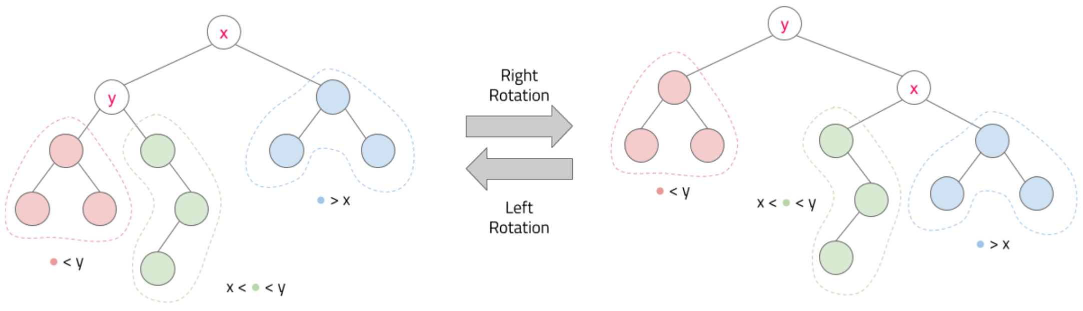

Figure 5: The right and left rotations explained without words. It's easy to see that the new tree isn't just an isomorphic variant (the node now is directly connected to a green node).

Both operations are common in self-balancing BST, like AVL and red black trees. Each data structure uses different strategies to identify the best moment and the best place to rotate the tree. We'll, on the other hand, use rotations in a different way.

To insert a node on a treap, we'll, in a first moment, ignore the priority and insert it like a simple BST.

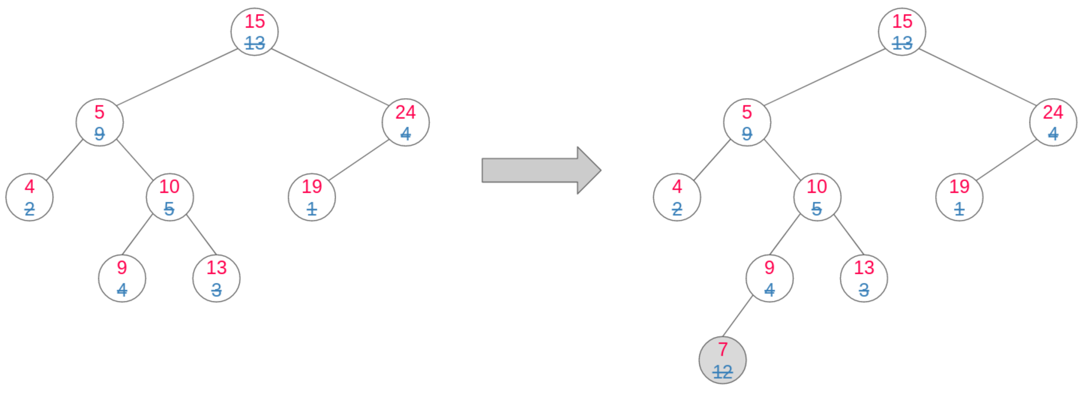

Figure 6: The first step to insert a node in a treap is to ignore the priorities and insert it like a BST. In this example, we inserted the node (7, 12).

Even though BST property is being preserved, the heap property is not. We should move the new node (7, 12) up, but if we do that the BST property will break. How can we fix that?

We'll move it up using the rotations! The steps are described in figure 7.

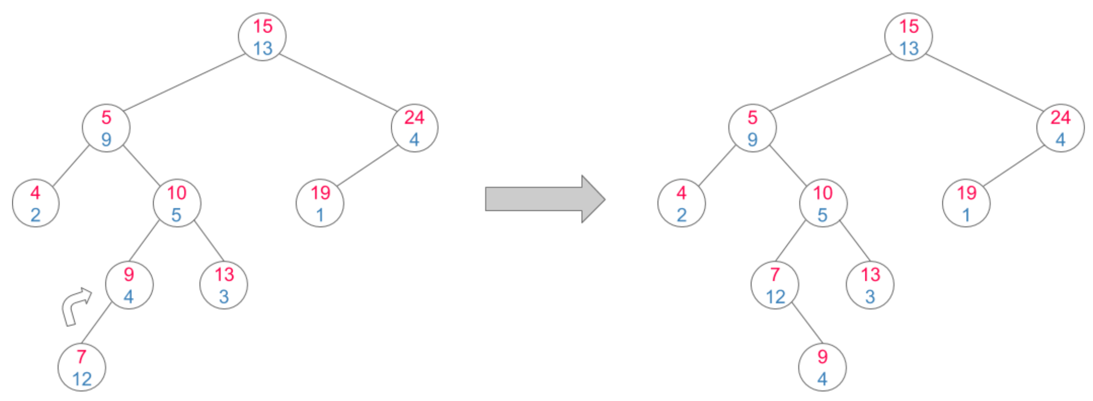

Figure 7a: The node (9, 4) is the father of (7, 12), but it doesn't have a higher priority. To fix that we must make a right rotation.

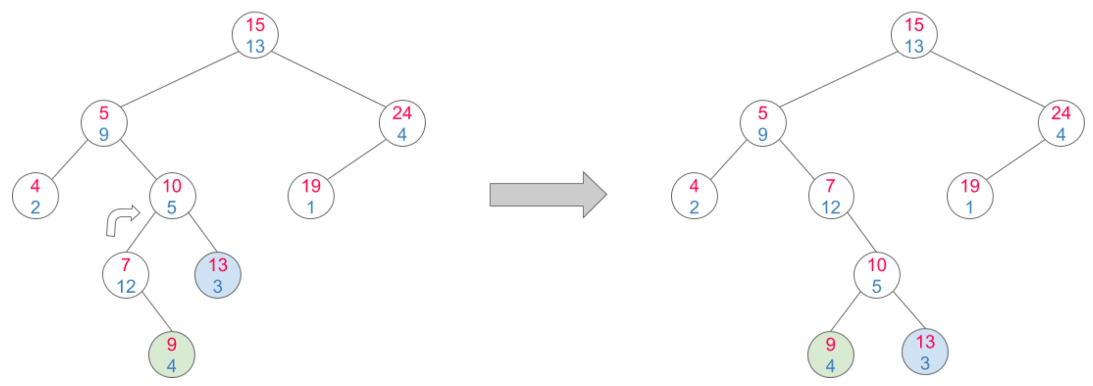

Figure 7b: The node (10, 5) is the father of (7, 12), but it doesn't have a higher priority. To fix that we must make another right rotation.

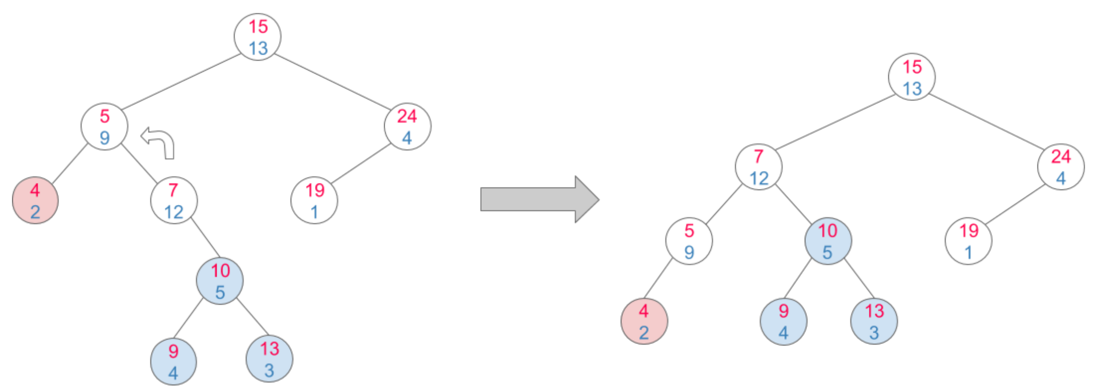

Figure 7c: The node (5, 9) is the father of (7, 12), but it doesn't have a higher priority. To fix that we must make another rotation, but a left one this time.

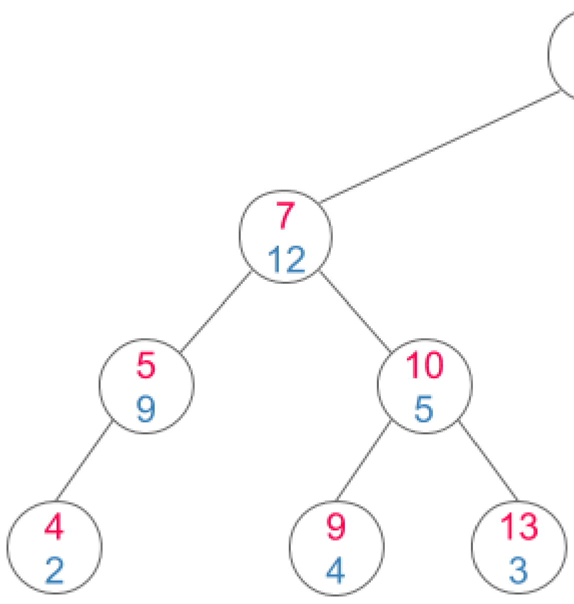

Figure 7d: The treap after all rotations.

In the first step, we go down from the root to the leaf. In the second step, we go up from a leaf to the root at maximum using O(1) rotations. It means that the insertion time complexity is O(2h) = O(log(n)).

We can understand a treap insertion as a BST insertion (considering only the key) followed by a heap insertion (start as a leaf and go up to the tree until you fix the heap property), which is another way to thing about the O(log(n)) time complexity.

Let's think a little more about the second step, when we start at a leaf and go up to the tree. The only way to go from the leaf to the root is if the new priority has the highest value. In other words, if you are generating random numbers using an uniform distribution, the new value must be the highest from a set of |T| (number of nodes of the tree), witch will happen with a probability of 1/|T| (try to prove that). It means that the rotations occur often near the leaves.

Deletion

Given a key, we can't just find its node and remove it. Pulling out internal nodes splits the tree and we don't want it. The solution is to move the node down to the tree and remove it only when it becomes a leaf.

The question now is what path do we choose to go down? That node may have two children, which one should we select?

To maintain the heap property, it’s easy to see that we have to rotate the tree from the children with higher priority (see image 10).

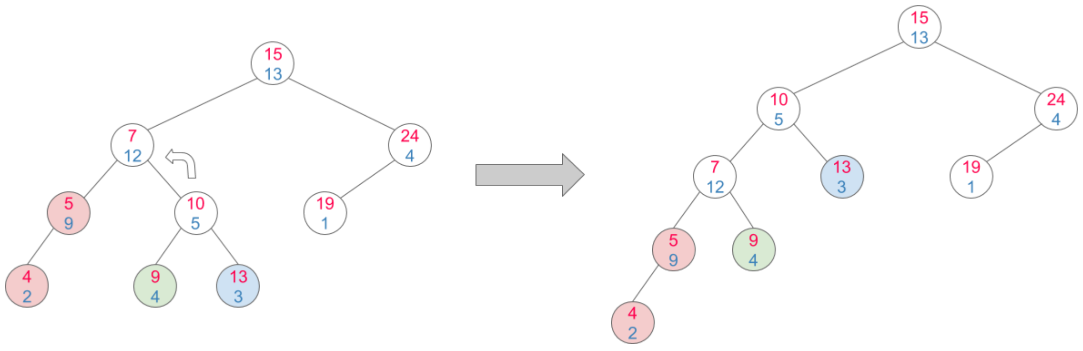

Figure 10a: We want to lower the node (7, 12), but if we rotate it with the children with lowest priority, we’ll break the heap property of the treap because the node (10, 5) will be above the node (5, 9).

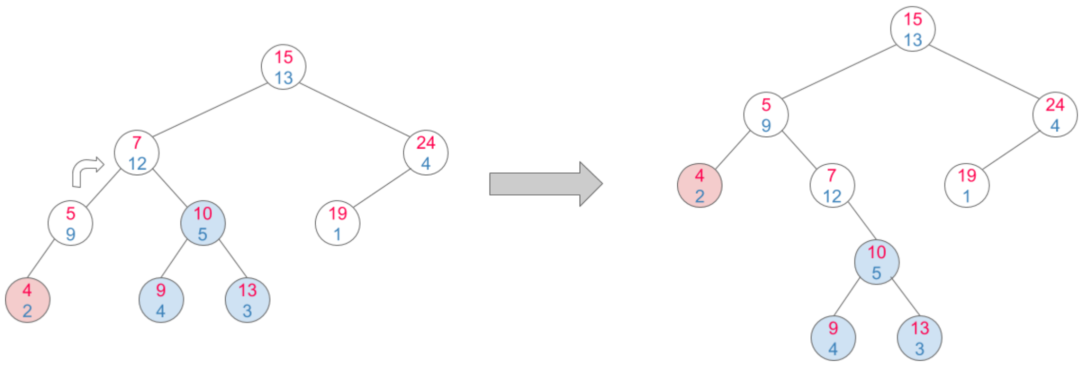

Figure 10b: If we choose to rotate with the children with highest priority, on the other hand, we’ll maintain the heap property. Note that there is no problem in nothe (5, 9) being above the node (10, 5).

The idea is simple: After the rotation, one of the children will be the father of the other one, so we have to “promote” the children with highest priority, otherwise we’ll break the heap property.