What is a treap?

A type of balanced binary search tree, easy to implement (I hope so). It is similar to binary search tree with randomized insertion order so average depth will be , but it can still retain its randomness with fixed order of insertions.

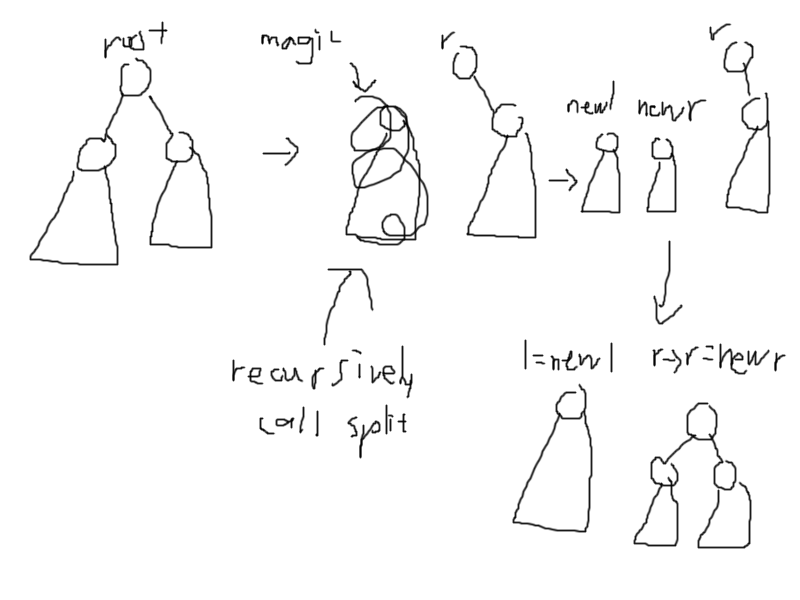
Every node has 2 value, key and priority. Key is like binary search tree, where the left child is smaller than the node and the right child is larger than the node. Priority is like a min-heap, where the priorities of both children are less than the priority of the node. (Hence the name treap.)

When using the data structure, the priority is randomly generated so that the average depth will be , and the time complexities of operations including insertion, deletion, and binary search are all .

Struct

The structure of a node will have 2 integers key and priority, and 2 node pointers left and right.

Split

The function have 2 inputs root and key, and 2 addresses (which can be thought of as outputs) left and right.

if the key is less than the key of the root, than it should be in the left subtree. You can just treat the function as a black box, and apply it to the left subtree. Now you have the new left, which is all smaller than the key (because that’s what split does) and the new right, which is all larger than the key and also smaller than the r in the diagram (because it was originally part of the left subtree and r is the right subtree). All you need to do is stick the new right to the left subtree of right, and you are done! This all seems very complicated, but it’s actually extremely easy to implement.

Also remember to check if the root is pointing to null, then just assign l and r as null.

Insert

The function should have root and the node you want to insert as inputs, and output the final pointer to the new root.

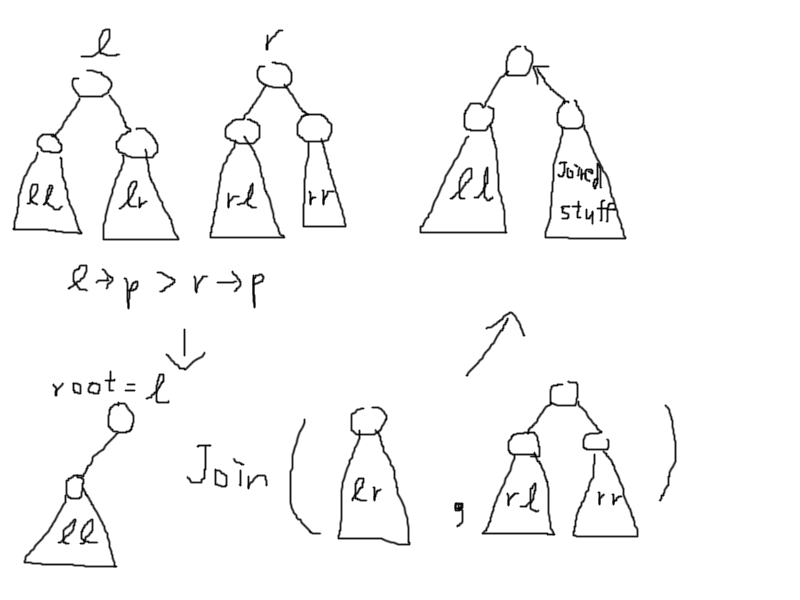
First check if the priority of the new node is larger than the priority of the root. If it is, the new node should be the new root, so just call split with the key of the node you want to insert, and return node!

If it is not larger, check if the new node belongs in the left or right subtree of root (by comparing their keys). Recursively call insert on that subtree, then return the root with now updated left or right subtree.

Join

The join function joins 2 treaps together, and one of them (left) is smaller (the keys are smaller) than the other (right).

There are 2 input pointers left and right, and 1 address pointer (pointer address? I still don’t know) root, which can be thought of as the output.

First, check if one of left or right is null. Then, compare the priorities of root of left subtree and root of right subtree. The priorities needs to maintain a max-heap property, and the priority of a node needs to be larger than or equal to the priorities of children. So, let’s say, the priority of the left subtree is greater, so the root of the left subtree should be the new root. Now, just merge the right subtree of the left subtree and the right subtree, then stick it in. Again, you can think of the join function as a black box, and it will magically work.

Erase

The function consists of (pass by reference??) root, and the key of the node you want to delete.

If key of the root is equal to the key, just merge the left and right subtree and update the root. Else, compare the key and the key of the root, and just like dfs down.

Implicit treaps

Treap is a data structure like a set, and you can do insertion, deletion and binary searches in times. Implicit treaps are like treaps but there is order. It is like a super-array. The inorder traversal of the treap is the array. It can also be a string, and the data structure is called a rope (dope name). The structure doesn’t have a key, but have a value, which is the element in the array, and a size, which is the subtree size.

With implicit treaps, you can insert element according to the index (it doesn’t need to be sorted), erase elements according to the index, split, join, function over an interval (such as sum, minimum, etc.), lazy propagation (such as add delta to a specific interval), and reverse an interval (extremely cool) all in time.

Tutorials on internet says something like instead of using the values as keys, implicit treaps use array indices as keys, but not explicitly, because that would be slow, blah blah blah. Based on my understanding, treap is tree + heap, and implicit treap is array + heap, where the binary search tree property of keys are changes to that the inorder traversal of keys form the array. The index of a node in the array is calculated on the fly using subtree sizes.

https://blog.csdn.net/CABI\_ZGX/article/details/79963427