Randomized Search Trees

Step 1

In the previous section of this chapter, we were able to formally prove that, after making two assumptions about randomness, the average-case time complexity of **BST** insertion, removal, and finding is O(log *n*). However, we also concluded that, in practice, those two assumptions we made were pretty unrealistic. Can we salvage all of the hard work we just did? Is there some clever way for us to simulate the same random distribution of tree topologies? If, in practice, we are somehow able to successfully simulate the same exact randomness in tree structure that we assumed in our proof, we would be able to actually experience this average-case time complexity.

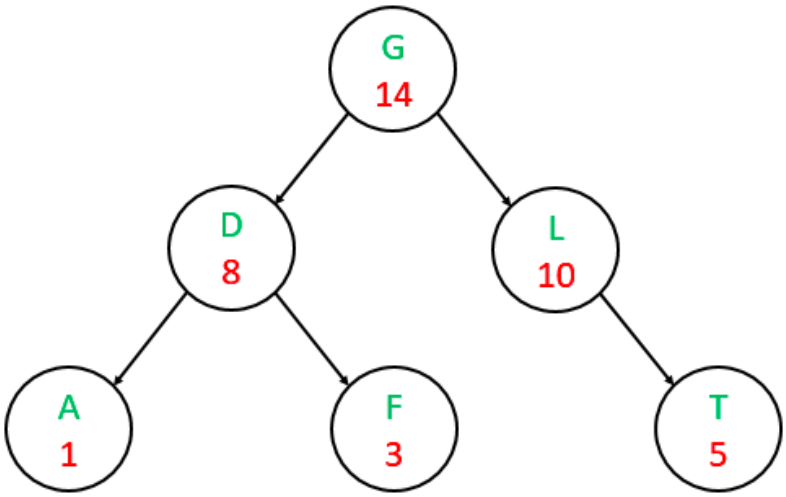
It turns out that, by tying in the ***Heap Property*** (from the previous section on Heaps) with some random number generation to our **Binary Search Trees**, we can actually achieve this simulation, and as a result, the O(log *n*) average-case time complexity. The data structure we will be discussing in this section is the **Randomized Search Tree (RST)**.

Step 2

The **Randomized Search Tree (RST)** is a special type of **Binary Search Tree** called a **Treap** ("Tree" + "Heap"). Formally, a **Treap** is a binary tree in which nodes contain two items, a *key* and a *priority*, and which must obey the following two restrictions:

1. The tree must follow the **Binary Search Tree** properties with respect to its **keys**
2. The tree must follow the ***Heap Property*** with respect to its **priorities**

Below is an example of a valid **Treap**, where **keys** are **letters** (with alphabetic ordering) and **priorities** are **integers** (with numeric ordering):



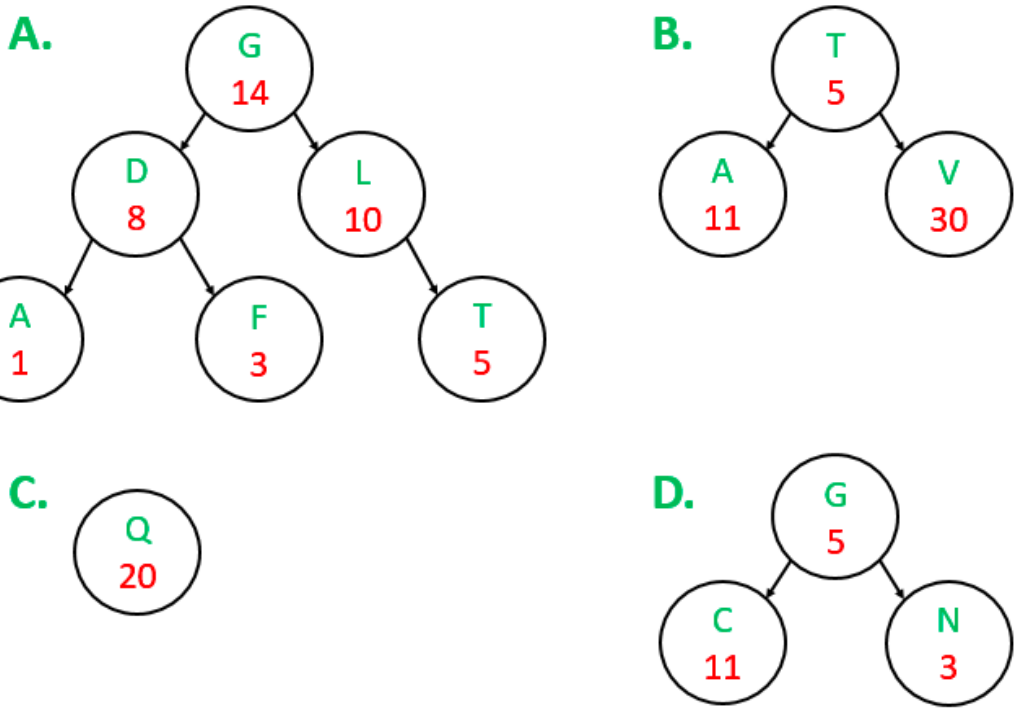
Given a set of (*key*, *priority*) pairs, where all keys are unique, we can easily construct a valid **Treap** containing the given pairs:

- Start with an empty **Binary Search Tree**
- Insert the (*key*, *priority*) pairs in decreasing order of *priority*, using the regular **Binary Search Tree** insertion algorithm with respect to the *keys*
- The resulting **BST** is a **Treap**: the **BST** ordering of the *keys* is enforced by the **BST** insertion algorithm, and the ***Heap Property*** of the *priorities* is enforced by inserting pairs in descending order of priorities

However, this trivial algorithm to construct a **Treap** relies heavily on knowing in advance *all* of the elements so that we can sort them and insert them all at once. Instead of making this unrealistic assumption, let's explore algorithms that allow for more realistic dynamic use of the structure.

Step 3

EXERCISE BREAK: Which of the following are valid **Treaps**? (Select all that apply)

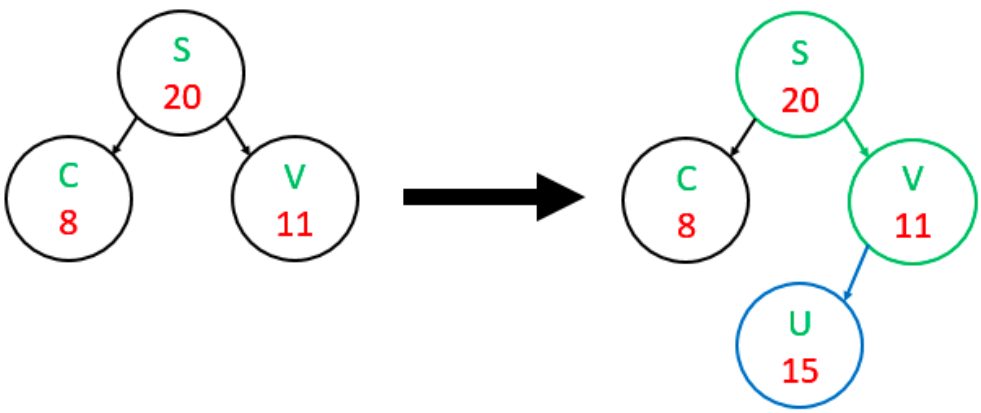


To solve this problem please visit https://stepik.org/lesson/28864/step/3

Step 4

Because a **Treap** is just a type of **Binary Search Tree**, the algorithm to find a *key* in a **Treap** is completely identical to the "find" algorithm of a typical **BST**: start at the root node, and then traverse left or right down the tree until you either find the desired element (or fall off the tree if the desired element doesn't exist). The **Binary Search Tree** "find" algorithm is extensively described in the **Binary Search Tree** section of this chapter, so if you're having trouble remembering it, be sure to reread that section.

Inserting a new (*key*, *priority*) element into a **Treap** is a bit trickier. The first step is to insert the (*key*, *priority*) pair using the typical **BST** insertion algorithm (using the *key* of the pair). After the **BST** insertion, the resulting tree will be valid with respect to *key* ordering, but because we ignored the *priority* of the pair, there is a strong chance that the resulting tree will violate the ***Heap Property***. To clarify this realization, below is an example of a **Treap** just after the **BST** insertion algorithm, where the element inserted is **(U, 15)**:

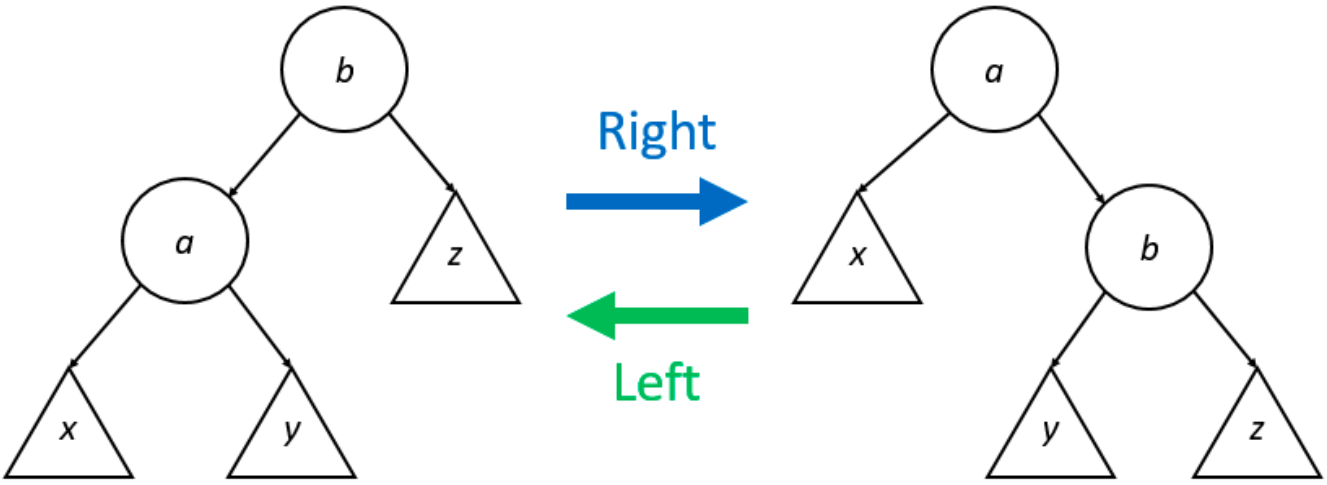


Notice how the **Binary Search Tree** properties are maintained with respect to the *keys*, but we have now violated the ***Heap Property*** with respect to the *priorities* (15 is larger than 11). Recall from the **Heap** section of the text that we typically fix the ***Heap Property*** by bubbling up the newly-inserted element. Is there some way for us to bubble up the new element to fix the ***Heap Property***, but without breaking the **Binary Search Tree** properties?

Step 5

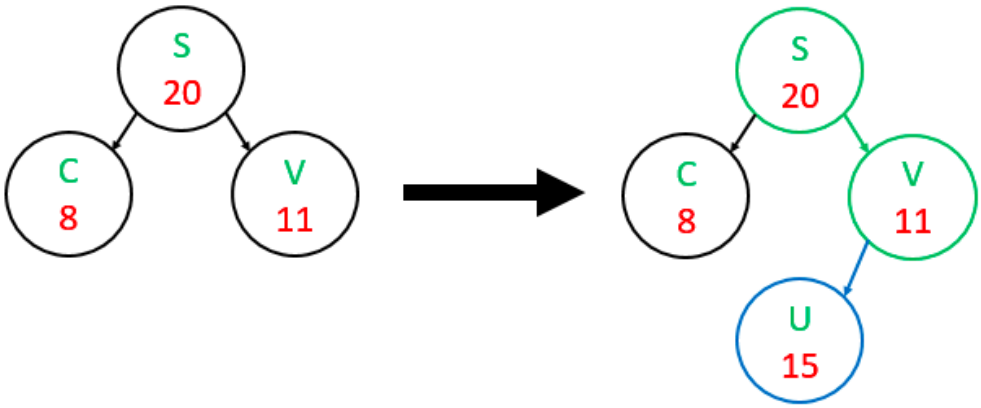
It turns out that we actually are able to "bubble up" a given node without destroying the **Binary Search Tree** properties of the tree via an operation called an **AVL Rotation**. This operation was originally designed for the "AVL Tree" data structure, which we will cover extensively in a later section of the text, but we will borrow the operation for use with our **Treaps**.

AVL Rotations can be done in two directions: **right** or **left**. Below is a diagram generalizing both right and left **AVL Rotations**. In the diagram, the triangles represent arbitrary subtrees of any shape: they can be empty, small, large, etc. The circles are the "important" nodes upon which we are performing the rotation.

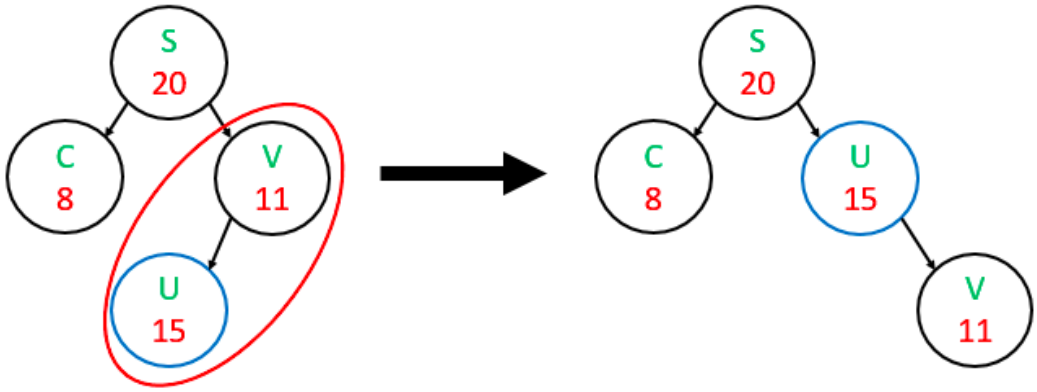


Step 6

Recall that we attempted to insert the element **(U, 15)** into a **Treap**, which disrupted the ***Heap Property*** of the tree (with respect to *priorities*):



Now, we will use **AVL Rotations** to bubble up the new element (in order to fix the ***Heap Property***) without disrupting the **Binary Search Tree** properties:



As you can see, we succeeded! We were able to bubble up the node that was violating the ***Heap Property*** in a clever fashion such that we maintained the **Binary Search Tree** properties. The first step of the **Treap** insertion algorithm was the regular **Binary Search Tree** insertion algorithm, which is O(*h*), where *h* is the height of the tree. Then, the bubble up AVL rotation step performs a single rotation (which is a constant-time operation) at each level of the tree on which it operates, so in the worst case, it would perform O(*h*) AVL rotations, resulting in a worst-case time complexity of O(*h*) to perform the AVL rotation step. As a result, the overall time complexity of the **Treap** insertion is **O(*h*)**.

Step 7

Below is formal pseudocode for each of the three **Treap** operations: find, insert, and remove. Note that we did not explicitly discuss **Treap** removal, but just like with **Treap** insertion, the first step is to simply perform **BST** removal, and if we have broken the ***Heap Property***, fix it with AVL rotations (to either bubble up or trickle down the node in violation).

If you're unsure of any of the **BST** algorithms referenced in any of the following pseudocode, reread the **BST** section of this text to refresh your memory.

```
find(key):
    perform BST find based on key
```

```
insert(key, priority):
    // Step 1: BST Insertion Algorithm
    node = (key, priority)
    perform BST insertion based on key

    // Step 2: Fix Heap Property via Bubble Up
    while node is not root and node.priority > node.parent.priority:
        if node is left child of node.parent:
            perform right AVL rotation on node.parent
        else:
            perform left AVL rotation on node.parent
```

```
remove(key):
    // Step 1: BST Removal Algorithm
    perform BST removal based on key
    node = node that was moved as a result of BST removal (i.e., the successor of key)
    if Heap Property is not violated:
        return

    // Step 2: Fix Heap Property if necessary
    // Bubble Up if necessary
    while node is not root and node.priority > node.parent.priority:
        if node is left child of node.parent:
            perform right AVL rotation on node.parent
        else:
            perform left AVL rotation on node.parent

    // Otherwise, Trickle Down if necessary
    while node is not a leaf and node.priority < either of its children's priorities:
        if node.leftChild.priority < node.rightChild.priority:
            perform left AVL rotation on node
        else:
            perform right AVL rotation on node
```

Step 8

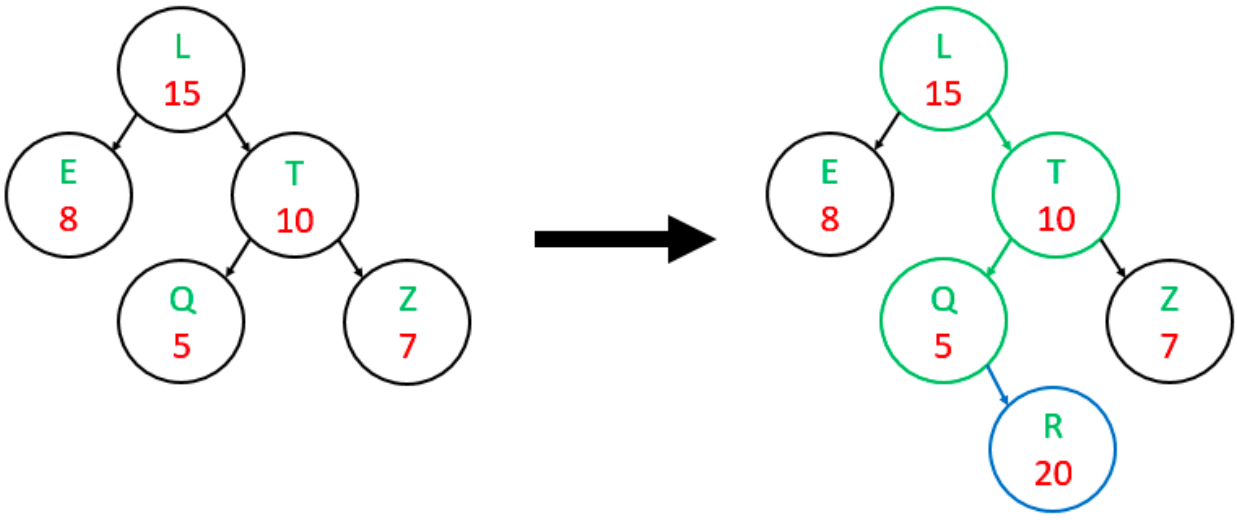
EXERCISE BREAK: Previously, we described the time complexity of insertion into a **Treap** with respect to the tree's height, *h*. With respect to the number of elements in the tree, *n*, what is the worst-case time complexity of **Treap** insertion?

To solve this problem please visit https://stepik.org/lesson/28864/step/8

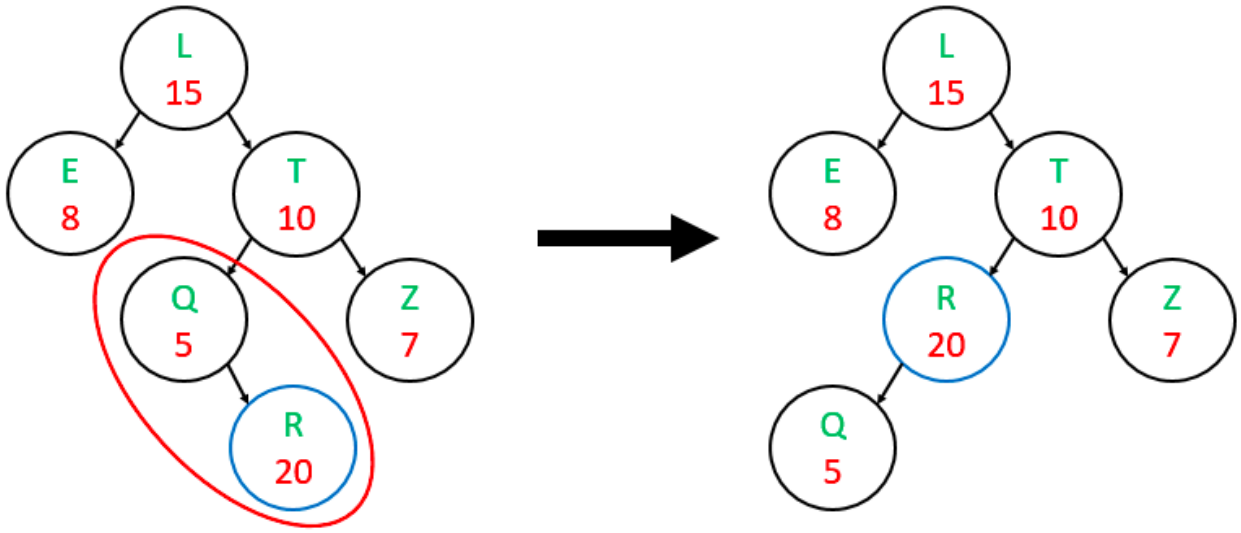
Step 9

Hopefully the question about worst-case time complexity on the previous step jolted your memory a bit: remember that we are jumping through all of these hoops in order to obtain a O(log *n*) *average-case* time complexity, but the *worst-case* time complexity remains unchanged at O(*n*) if the tree is very unbalanced.

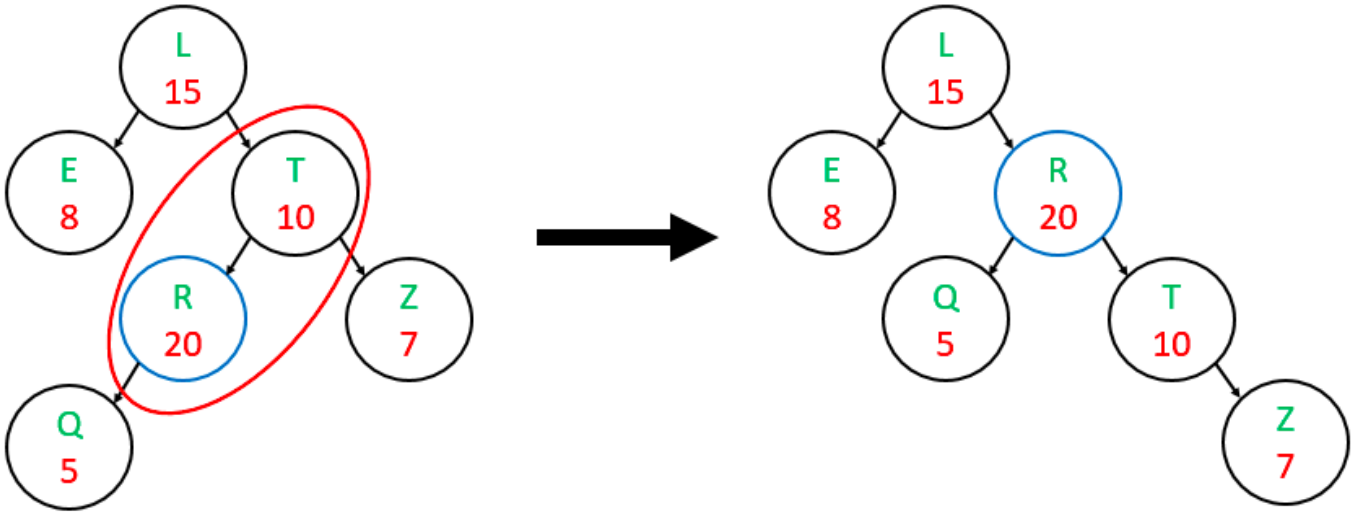
Nevertheless, below is a more complex example of insertion into a **Treap**. In the example below, the element being inserted is **(R, 20)**:



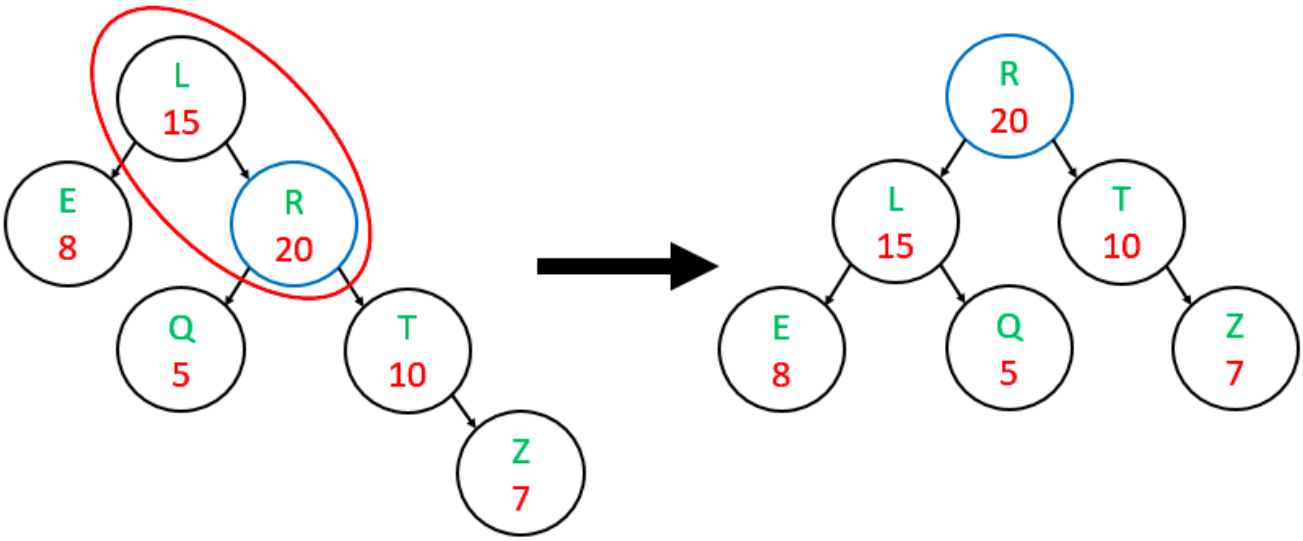
Now that we've done the naive **Binary Search Tree** insertion algorithm, we need to bubble up to fix the ***Heap Property***. First, we see that the new element's priority, 20, is larger than its parent's priority, 5, so we need to perform an AVL rotation (a left rotation) to fix this:



Next, we see that the new element's priority, 20, is larger than its parent's priority, 10, so we need to again perform an AVL rotation (a right rotation) to fix this:



Finally, we see that the new element's priority, 20, is larger than its parent's priority, 15, so we need to perform one final AVL rotation (a left rotation) to fix this:



Again, success! We were able to insert our new (*key*, *priority*) element into our **Treap** in a manner that maintained the **Binary Search Tree** properties (with respect to *keys*) as well as the ***Heap Property*** (with respect to *priorities*). Also, hopefully you noticed something interesting: after the initial naive **Binary Search Tree** insertion step of the **Treap** insertion algorithm, the resulting tree was quite out of balance, but by chance, because of the new element's *priority*, the bubble-up step of the **Treap** insertion algorithm actually improved our tree's balance! This peculiar phenomenon motivates the data structure we introduced at the beginning of this section: the **Randomized Search Tree**.

Step 10

A **Randomized Search Tree** is a **Treap** where, instead of us supplying both a *key* and a *priority* for insertion, we only supply a *key*, and the *priority* for the new (*key*, *priority*) pair is randomly generated for us.

Given that a **Randomized Search Tree** is just a **Treap**, the pseudocode for its find, insert, and remove operations should seem very trivial:

```
find(key):
    perform BST find based on key
```

```
insert(key):
    priority = randomly generated integer
    perform Treap insert based on (key, priority)
```

```
remove(key):
    perform Treap remove based on key
```

Recall that we began this exploration of **Treaps** and **Randomized Search Trees** because we wanted to obtain the O(log *n*) average-case **Binary Search Tree** time complexity we formally proved in the previous section of this text without having to make the two unrealistic assumptions we were forced to make in the proof. It turns out that, via a formal proof that is beyond the scope of this text, by simply randomly generating a *priority* for each *key* upon its insertion into our tree, we are able to in-practice (i.e., with real data) simulate the exact same random distribution of tree structures that we needed, meaning we have actually succeeded in achieving an average-case time complexity of **O(log *n*)**!

You might be thinking "Wait... That's it?", and you're absolutely right: that's it. To reiterate in the event that it didn't quite register, we have simulated the random tree topologies needed to achieve the O(log *n*) average-case time complexity of a **BST** by simply creating an **RST**, which is a **Treap** in which the *keys* are the same as we would be inserting into a **BST**, but the *priorities* are randomly generated.

Step 11

EXERCISE BREAK: With respect to the number of elements in the tree, *n*, what is the worst-case time complexity of the **Randomized Search Tree** find operation?

To solve this problem please visit https://stepik.org/lesson/28864/step/11

Step 12

EXERCISE BREAK: With respect to the number of elements in the tree, *n*, what is the average-case time complexity of the **Randomized Search Tree** find operation?

To solve this problem please visit https://stepik.org/lesson/28864/step/12

Step 13

Throughout this lesson, we were able to design a new data structure, the **Randomized Search Tree**, that is able to exploit the functionality of a **Treap** with random number generation in order to achieve the theoretical **O(log *n*) average-case time complexity** *in practice* for the **Binary Search Tree** operations.

However, hopefully you noticed (either on your own, or via the trick questions we placed throughout this section) that, although we've obtained an average-case time complexity of O(log *n*) by clever trickery, we still face our existing worst-case time complexity O(*n*). Is there any way we can be even *more* clever and bump the worst-case time complexity down to O(log *n*) as well? In the next section, we will discuss the first of two tree structures that actually achieves this feat: the **AVL Tree**.