Exploration: Rotation Implementation

Introduction

We have looked at when we need to make rotations and identified situations where we might need to make multiple rotations. This exploration will look at the implementation of these rotations in code.

Single Rotations

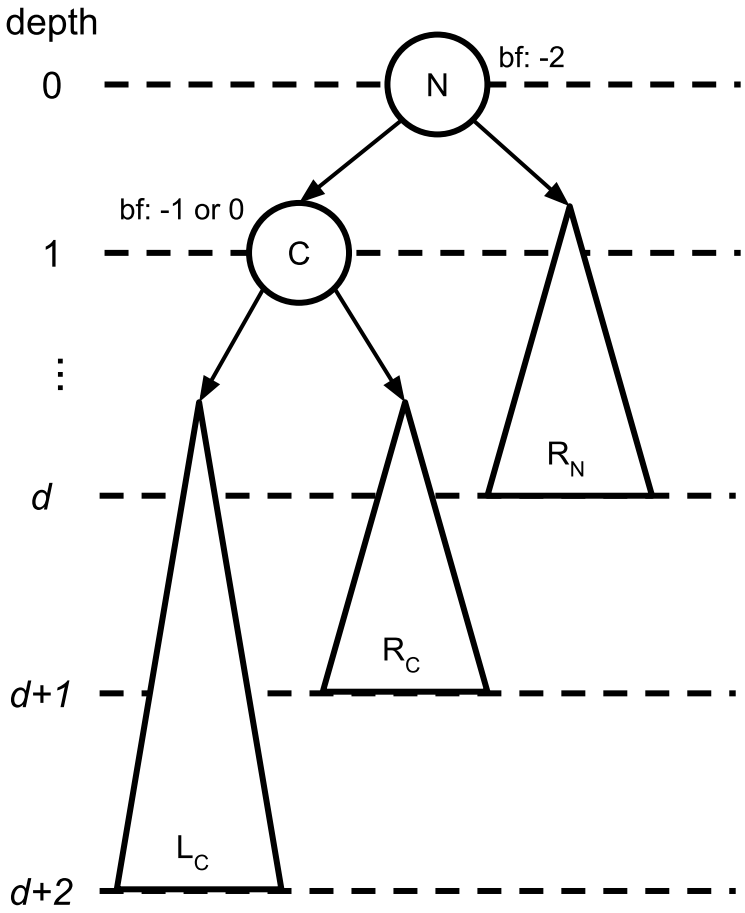
Recall from above that a single rotation is needed to restore height balance at a node N in situations where N's child C is heavy in the same direction as N. In other words, a single rotation is needed in the following situations, where N is the node at which height balance is lost:
- Left-left imbalance – N is left-heavy and N's left child C is also left-heavy
- Right-right imbalance – N is right-heavy and N's right child C is also right-heavy

Again, a single rotation can be either a left rotation or a right rotation. When applying a single rotation, it is always centered around the node N where height balance is lost, and the rotation is in the opposite direction of the imbalance, i.e.,:

- To fix a left-left imbalance at N, we apply a single right rotation around N.
- To fix a right-right imbalance at N, we apply a single left rotation around N.

Here, we'll examine what a single right rotation looks like visually. Just remember that a single left rotation would simply mirror a right rotation.

Again, a single right rotation is needed to restore height balance when there's a left-left imbalance at a node N (i.e., N's balance factor is –2 and the balance factor of N's left child C is –1 or 0). This could occur after an element is inserted into C's left subtree, causing it to grow in height by 1, or it could occur after a removal from N's right subtree, causing it to shrink in height by 1. This is depicted below:



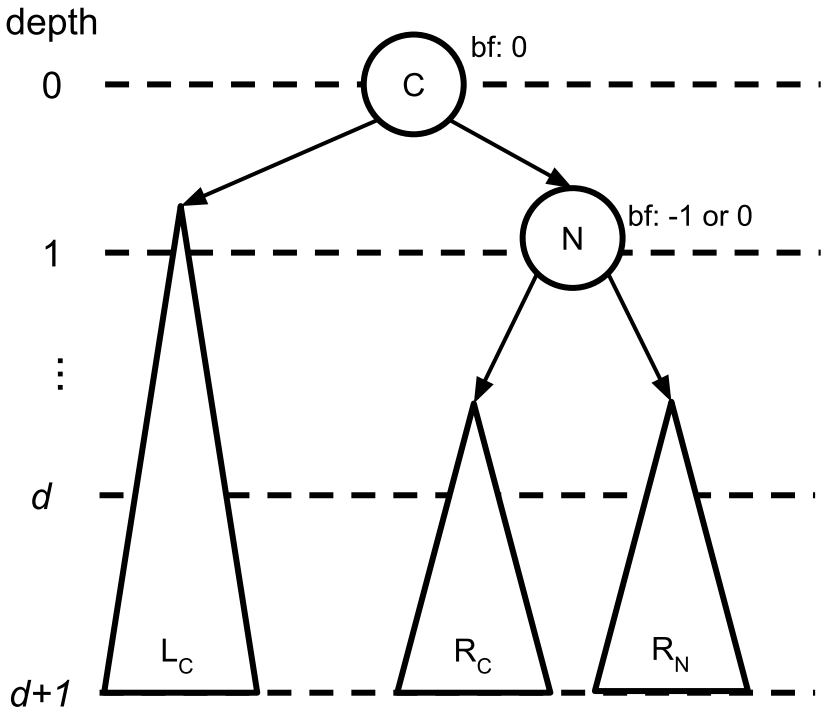
A tree requiring a single right rotation to fix the imbalance

Note here that the exact heights of the subtrees LC, RC, and RN are not important, except insofar as they contribute to the balance factors of N and C. Specifically, what really matters here is that N has a balance factor of –2 and that C has a balance factor of –1 or 0. Again, the scenario here is specifically set up so that we'll perform a right rotation. The scenario would be mirrored if we were doing a left rotation.

In a right rotation around N, the following things will happen:

- N will become the right child of its current left child C.
- C's current right child will become N's left child.
- If N has a parent PN, then C will replace N as PN's child. Otherwise, if N was the root of the entire tree, C will replace N as the root.

Visually, here's how the restructured subtree originally rooted at N would look after this right rotation:



A now balanced tree after a right rotation

As desired, the rotation reestablishes height balance here. Importantly, note again that the rotation results in its center N moving downward in the tree, and N's original child C moving upward to become the new root of the subtree. Again, if we were doing a left rotation, the operation above would be mirrored.

Double Rotations

Again, recall from above that a double rotation is needed to restore height balance at a node N in situations where N's child C is heavy in the opposite direction as N. In other words, a double rotation is needed in the following situations, where N is the node at which height balance is lost:

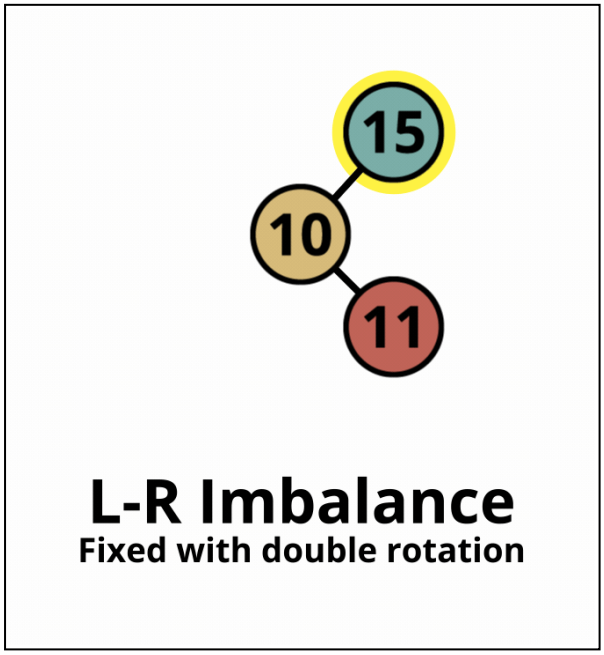
- **Left-right imbalance** – N is left-heavy and N's left child C is right-heavy
- **Right-left imbalance** – N is right-heavy and N's right child C is left-heavy

As the name implies, a double rotation consists of two single rotations. The first of these rotations is always centered around N's child C, and the second is always centered around N itself (where, as before, N is the node where height balance is lost).

Importantly, each rotation in a double rotation is always applied in the direction opposite to the heaviness at the rotation's center. In other words:

- To fix a left-right imbalance, we apply a left rotation around C followed by a right rotation around N.
- To fix a right-left imbalance, we apply a right rotation around C followed by a left rotation around N.

A very simple example of a double rotation is depicted below. This double rotation fixes a left-right imbalance. Specifically, in the example below, the node with key 15 loses height balance and is left-heavy, and its left child, the node with key 10, is right-heavy:



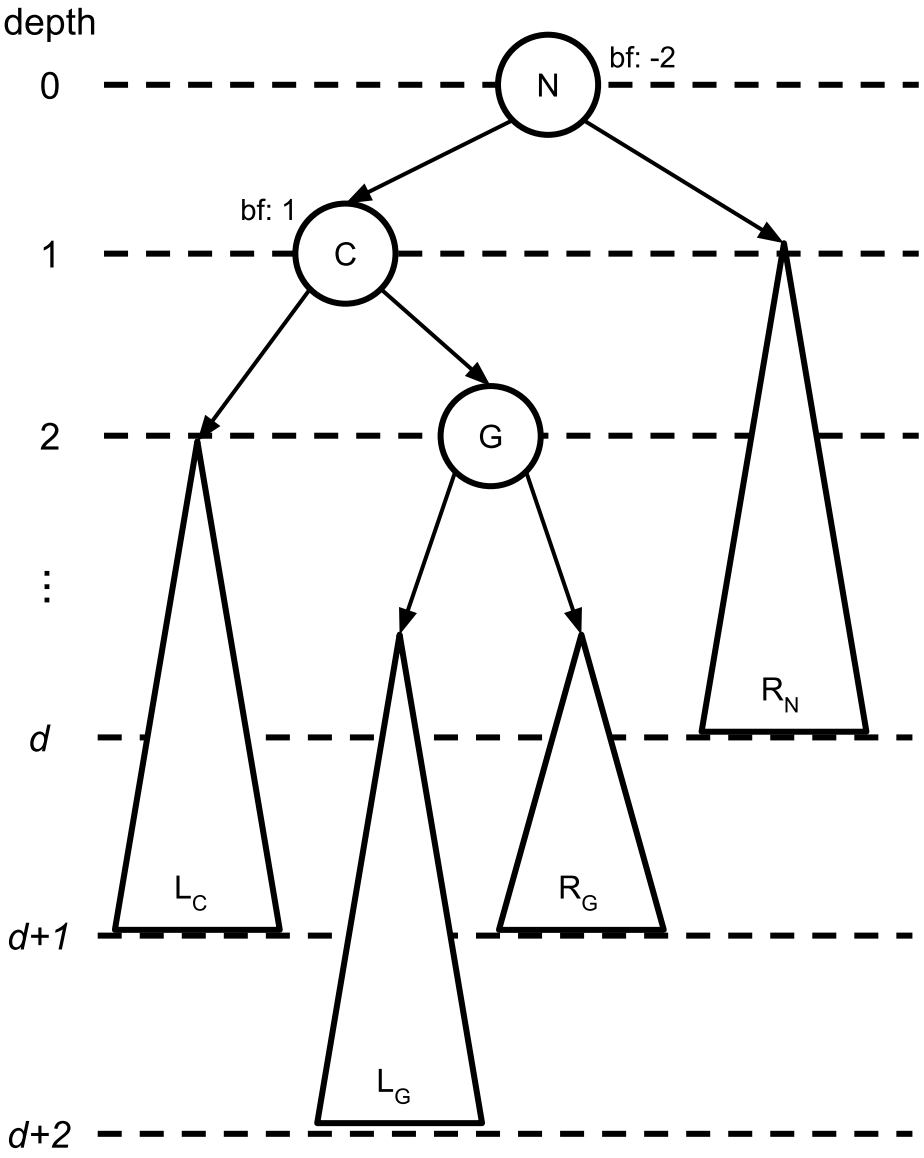
A tree being fixed with a double rotation

As the illustration above might make clear, the first rotation in a double rotation is simply intended to align imbalances on the same side, i.e., to create a left-left imbalance or a right-right imbalance. As we saw above, once imbalances are aligned in such a way, a single rotation is sufficient to restore height balance.

Let's walk through a more complete visual example of what a double rotation looks like. We'll again focus only on a double rotation to fix a left-right imbalance (i.e., a left rotation around C followed by a right rotation around N). As with the single rotation we explored above, the double rotation we see here would simply be mirrored in the case of a right-left imbalance.

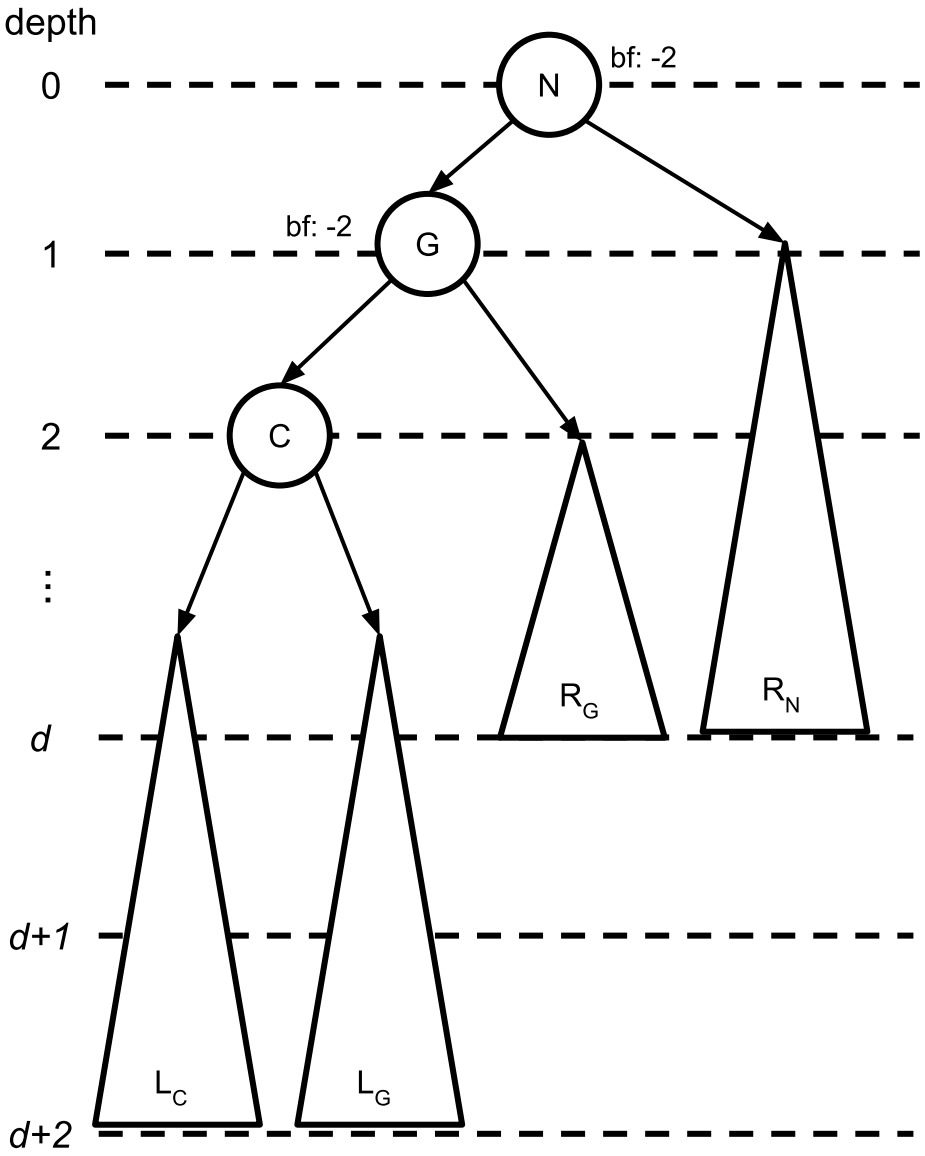
Again, the left-right imbalance we'll fix here will be indicated by node N having a balance factor of –2 (i.e., N has lost height balance and is left-heavy) and N's left child C having a balance factor of 1 (i.e., C is right-heavy). This situation could arise, for example, when an element is inserted into C's right subtree, causing its height to increase by 1, or it could arise when an element is removed from N's right subtree, causing its height to decrease by 1.

Visually, this imbalance would look as depicted below. Note that in this visualization, we're paying attention to an additional node G, which is C's right child (and N's grandchild):



A tree with a left heavy root and a right heavy left child

Again, we'll perform a double rotation to fix this left-right imbalance. The first rotation within the double rotation will be centered around C and in the opposite direction of C's imbalance, i.e., a left rotation:



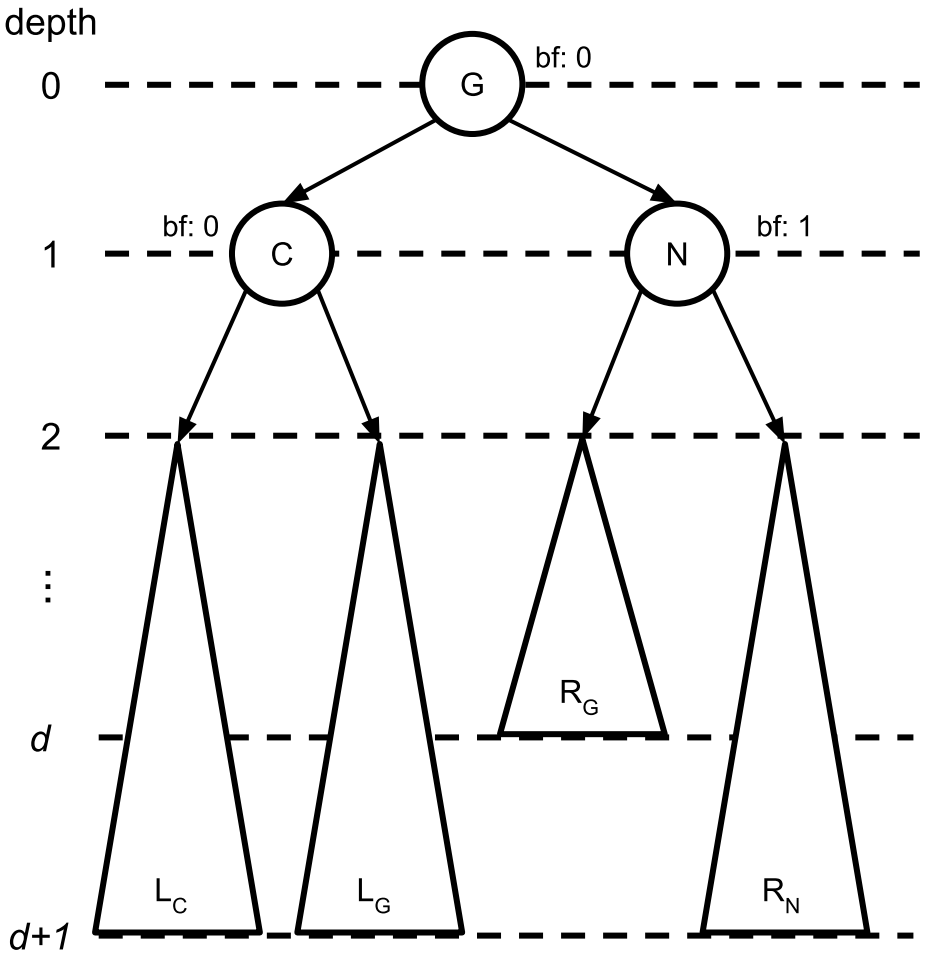
A tree that has had the left rotation about the left child completed

This first rotation works just like (the mirror of) the single rotation we examined above. Specifically:

1. G moves up in the tree to replace C as N's left child.
2. C moves down in the tree to become G's left child.
3. LG becomes C's right child.

The result here, again, is not that height balance is fixed, but simply that we've created a left-left imbalance (instead of the original left-right imbalance): N is left-heavy, and N's (new) left child G is also left heavy.

The second rotation of the double rotation will restore height balance. This second rotation will be centered around N, and as always, it will be in the direction opposite of N's imbalance, i.e., a right rotation around N:



A rebalanced tree after the right rotation about the root

Note here that G has moved up from its original position in the tree by two levels to become the new root of this subtree. As before, if N originally had a parent, PN, G would replace N as the child of PN after the double rotation. If N was originally the root of the entire AVL tree, G would become the new root.

Amazingly (or maybe not so amazingly, given that these rotation operations are precisely designed), the BST property still holds in this entire subtree after the double rotation is completed.

Look, for example, at the original subtrees of G, LG and RG. Though neither of these are still directly connected to G, they are still on the same sides of G. Similarly, G itself is still to the right of C and to the left of N.

Practical Implementation

The rotations we saw above are isolated in nature. They repair height balance within a single subtree of a larger AVL tree. This single repair, though, won't necessarily restore height balance within the larger AVL tree. Multiple rotations could be needed to completely restore height balance after a given insertion into or removal from an AVL tree.

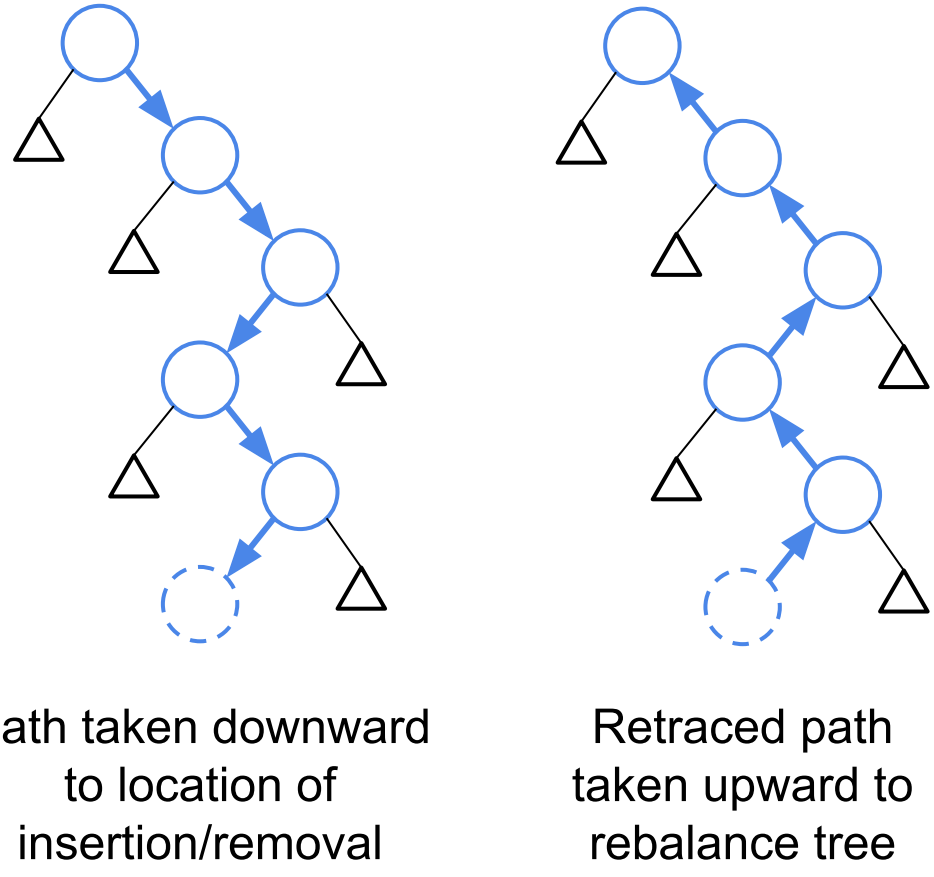
We also haven't yet seen the mechanics of how an AVL tree recognizes when and where a rotation is needed. For example, how does an AVL tree actually compute a node's balance factor, and how does an AVL tree recognize when a node's balance factor reaches –2 or 2 (i.e., when that node loses height balance)?

To clear all of this up, let's start to explore how the individual pieces we saw above fit together within the larger operation of an AVL tree.

The first thing we'll need to recognize is that an AVL tree will only ever need to be rebalanced in response to an operation that changes the structure of the tree, i.e., inserting a new element or removing an element. This means that we'll ultimately incorporate mechanics for tree rebalancing into these two operations.

Rebalancing an AVL tree is a bottom-up operation. In particular, rebalancing an AVL tree after an insertion or removal begins at the location in the tree where its structure was changed (i.e., at the spot where a node was inserted or removed) and proceeds upwards from that location towards the root.

Specifically, after every insertion into or removal from an AVL tree, the tree retraces the path it took to find the location at which to insert or remove a node upwards, towards the root. At each node encountered during this upward traversal of the tree, the AVL tree re-computes the balance factor and then rotates if needed:



The path taken back up a tree after checking balance after modifying a node

Importantly, this means that we'll need to introduce a mechanism to allow us to retrace a path upwards from a given node back to the root.

The simplest way to do this is by adding a pointer to the AVL tree node structure that points to the node's parent. Then, retracing the path upwards from a node to the tree's root is as simple as following these parent pointers up the tree.

While we're modifying the structure that represents each node in our AVL tree, we'll add an additional field that allows us to track the height of the subtree rooted at each node. If we were writing a Python **Node** class to represent each node we would need the following properties:

- **key** - an integer that the nodes are sorted by
- **value** - the value held by the node
- **height** - an integer representing the height of the node
- **left** - the left child *Node*
- **right** - the right child *Node*
- **parent** - the parent *Node*

Importantly, like we use `None` to indicate when a node doesn't have a child, we'll also use `None` to indicate when a node doesn't have a parent. Specifically, the root node of the tree will always have a `None` parent pointer.

Now that we've established how the tree will be represented, we'll more concretely specify how a rotation works. Here's pseudocode for a left rotation centered around a node N:

```
rotateLeft(N):
    C ← N.right
    N.right ← C.left
    if N.right is not NULL:
        N.right.parent ← N
    C.left ← N
    N.parent ← C
    updateHeight(N)
    updateHeight(C)
    return C
```

And here's the code for a right rotation around N:

```
rotateRight(N):
    C ← N.left
    N.left ← C.right
    if N.left is not NULL:
        N.left.parent ← N
    C.right ← N
    N.parent ← C
    updateHeight(N)
    updateHeight(C)
    return C
```

In both rotation functions, note that we return `C`, which has become the new root of the subtree at which the rotation was performed. We'll use this return value later to update `N`'s old parent to point to `C` (and vice versa).

In the rotation functions, we also use a function `updateHeight()`, which updates the height of a node whose subtrees may have been restructured:

```
updateHeight(N):
    N.height ← MAX(height(N.left), height(N.right)) + 1
```

Here, `N`'s new height is one more than the larger of the heights of its left and right subtrees (we add 1 to account for N itself). The `height()` function here simply returns a node's height or –1 if that node is NULL.

Just to review the way these pieces work:

- Rotating left or right around a given node works as described above and simply involves trading a few pointers.
- If we perform a rotation, we must re-compute the subtree heights for both the node that moved downwards during the rotation (i.e., N) and the node that moved upwards during the rotation (i.e., C).

With those pieces concretely specified, we can now incorporate restructuring functionality into the AVL tree's insert and remove operations. Here's pseudocode for the insert operation:

```
avlInsert(tree, key, value):
    insert key, value into tree like normal BST insertion
    N ← newly inserted node
    P ← N.parent
    while P is not NULL:
        rebalance(P)
        P ← P.parent
```

Similarly, here's pseudocode for the AVL tree's remove operation:

```
avlRemove(tree, key):
    remove key from tree like normal BST removal
    P ← lowest modified node (e.g. parent of removed node)
    while P is not NULL:
        rebalance(P)
        P ← P.parent
```

The key piece of both operations is the `rebalance()` function, which actually performs rebalancing at each node. Here's pseudocode for that function:

```
rebalance(N):
    if balanceFactor(N) < -1:
        if balanceFactor(N.left) > 0:
            N.left ← rotateLeft(N.left)
            N.left.parent ← N
        newSubtreeRoot ← rotateRight(N)
        newSubtreeRoot.parent ← N.parent
        N.parent.left or N.parent.right ← newSubtreeRoot
    else if balanceFactor(N) > 1:
        if balanceFactor(N.right) < 0:
            N.right ← rotateRight(N.right)
            N.right.parent ← N
        newSubtreeRoot ← rotateLeft(N)
        newSubtreeRoot.parent ← N.parent
        N.parent.left or N.parent.right ← newSubtreeRoot
    else:
        updateHeight(N)
```

Here are some highlights of this function:

- The if and else if conditions check if height balance is lost at N.
- The inner if statements check whether a double rotation is needed. Specifically, they check whether N's child is imbalanced in the opposite direction of N's imbalance. If so, the double rotation's first rotation, around N's child, is performed, with parent status updated appropriately.
- Any time a rotation is performed, the node returned by the rotation function is used to update parent status appropriately.
  - Note that newSubtreeRoot will become either the left or right child of N's old parent, replacing N.
- If no rotation at all is performed (i.e., if we enter the else block), we still need to make sure to update N's subtree height, since the tree underneath N may have changed.

Time Complexity

With the main AVL tree operations now concretely specified, we can assess the runtime complexity of those operations.

Let's start by focusing on a single rotation. It should be clear that a single rotation has constant (i.e., O(1)) time complexity. Specifically, here's what happens during a rotation:

- A limited (constant) number of pointers is updated (runs in constant time).
- The height of two nodes is updated (runs in constant time, since each update looks only at the heights of the node's two children).

Further, at most two rotations (i.e., a double rotation) will be performed during each call to `rebalance()`. Thus, `rebalance()` itself runs in constant time. How many times, then, will `rebalance()` be called?

Remember, `rebalance()` is called once per node on a traversal upwards to the root of the tree. It should be clear, then, that the maximum number of times `rebalance()` can be called is h (the height of the tree), which occurs when the upwards traversal starts at the very bottom of the tree.

As we saw above, if a tree is height balanced (like an AVL tree is), then its height is guaranteed to be within a constant factor of log(n). Thus, since `rebalance()` itself has O(1) complexity, the AVL tree's insert and remove operations each have overall complexity of O(log n).