# AVL Trees

## Self-balancing binary search trees

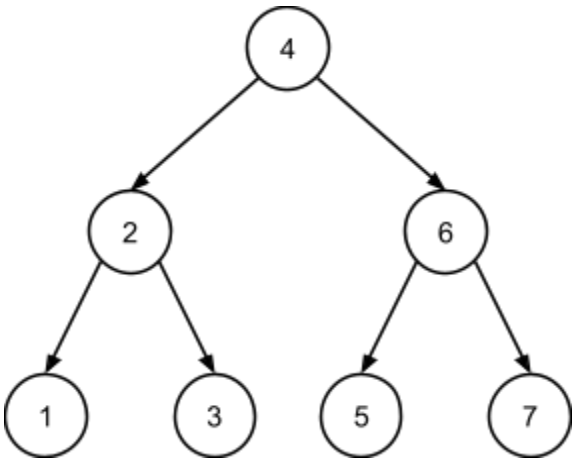
- The notion of ***balance*** is an important one when dealing with binary search trees (BSTs). While there are different ways to define the term “balance” itself, in general, when we talk about balance in the context of BSTs, we’re referring to trees in which all nodes have depth approximately *log n* or less.

- The reason balance is important when dealing with BSTs is because, as we’ve seen, the primary operations on BSTs all have *O(h)* runtime complexity, where *h* is the height of the tree.

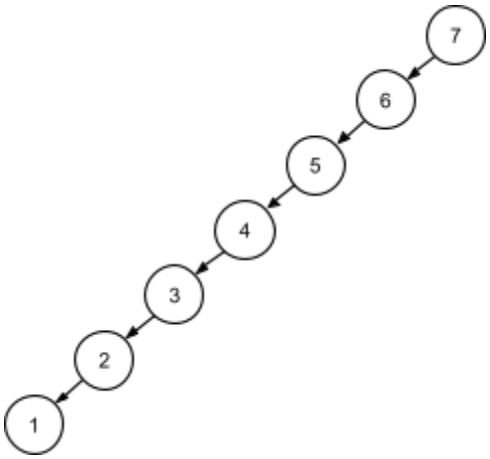
- When we’re working with a BST that’s more-or-less balanced, then *O(h)* operations will be fast, since in a balanced BST, *h* will be approximately *log(n)*. However, in a very *un*balanced BST, *h* will be closer to *n*, in which case operations on that BST could run much more slowly.

- Importantly, there’s nothing in the mechanics of plain BSTs to ensure that a tree is balanced. In particular, it’s important to realize that, for a given set of keys, the shape of a BST depends to a great degree on the order in which those keys are inserted into the tree.

- For example, if we insert elements with keys 1 through 7 into an empty BST in one particular order—4, 2, 6, 1, 3, 5, 7—the resulting tree will be perfectly balanced, resulting in operations with runtime around *O(log n)*:



- However, if we insert elements with those exact same keys into an empty BST in a different order—7, 6, 5, 4, 3, 2, 1—it results in a very *un*balanced tree, in which the runtime of operations will be closer to *O(n)*:




- Here, we'll begin to explore a type of data structure known as a ***self-balancing binary search tree***. As their name implies, a self-balancing BST does extra work (beyond that done by a plain BST) to try to ensure that the tree is more-or-less balanced as elements are inserted and removed.

- We'll specifically study a type of self-balancing BST known as an ***AVL tree***. First, though, we'll explore a more precise definition of "balance."

# Height balance and the balance factor

- ***Height balance*** is a measurable form of BST balance. Specifically, a BST is height balanced if, at every node in the tree, the subtree heights of the node's left and right subtrees differ by at most 1.

- Height balance is an important concept because it has been shown that a BST exhibiting height balance (i.e. one in which no node has subtrees that differ in height by more than 1) is guaranteed to have an overall height that's within a constant factor of *log(n)*.
    - In other words, because BST operations have *O(h)* runtime complexity, operations in a height-balanced BST are guaranteed to have *O(log n)* runtime complexity.

- A BST node's ***balance factor*** is a metric we can use to figure out whether the subtree rooted at that node is height balanced. Specifically, the balance factor of the node *N* is the height of *N*'s right subtree minus the height of *N*'s left subtree:

  **`balanceFactor(N) = height(N.right) - height(N.left)`**

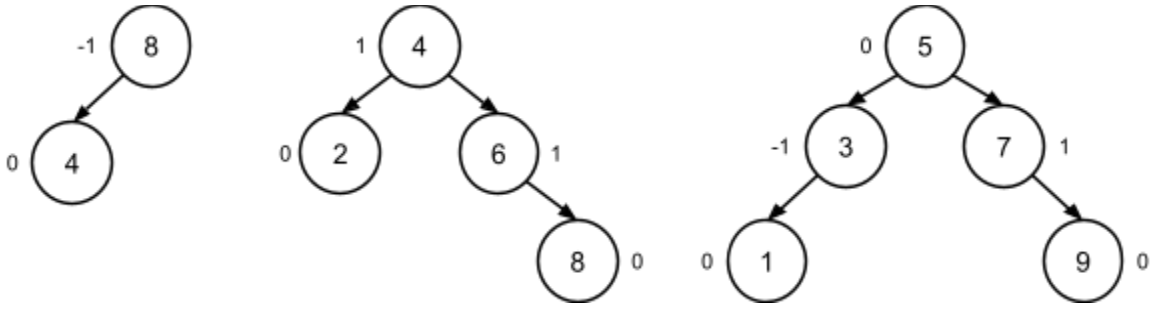
  - By convention, the height of a NULL node (i.e. an empty subtree) is -1.

- Balance factor and height balance are related concepts. Specifically, we can say that an entire BST is height balanced if every node in the tree has a balance factor of -1, 0, or 1.

- If a node has a negative balance factor (i.e. **`balanceFactor(N)`** < 0), we call it ***left-heavy***, since a negative balance factor implies that the node's left subtree has greater height than its right subtree.

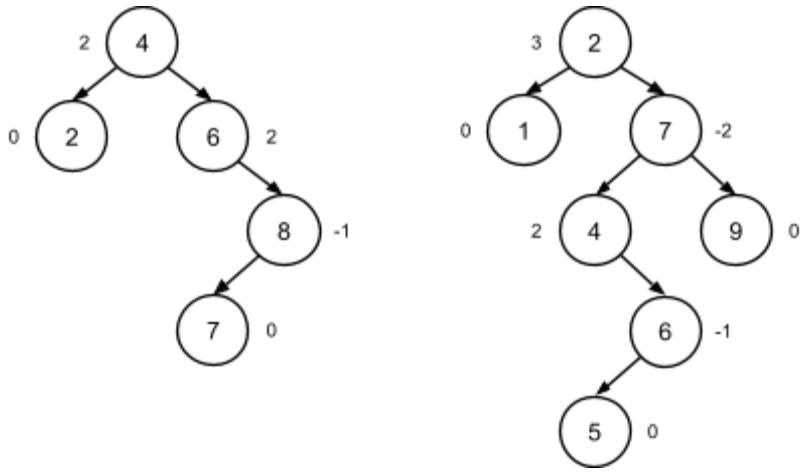
- Similarly, node whose balance factor is positive (i.e. **`balanceFactor(N)`** > 0) is called ***right-heavy***, since this node's right subtree would have greater height than its left subtree.

- Here are some examples of height-balanced BSTs, with the balance factor of each node denoted next to that node:




- Note that in the trees above, all nodes have a balance factor of either -1, 0, or 1, which, again, indicates that all of these trees are height balanced.

- And here are some examples of BSTs that are *not* height balanced, again, with the balance factor of each node denoted next to that node:




# Restructuring AVL trees via rotations

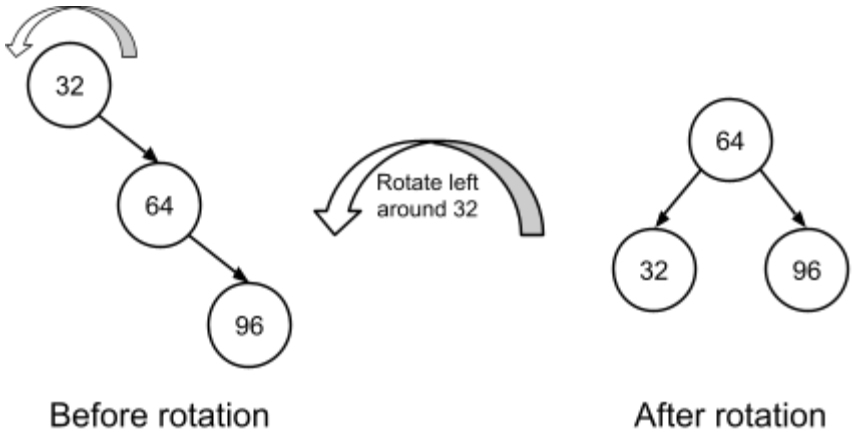
- The AVL tree is one of several existing types of self-balancing BST.
    - The acronym AVL is derived from the initials of the names of the tree's inventors: Adelson-Velsky and Landis.
    - There are many other kinds of self-balancing BST. Another popular one is the red-black tree.

- An AVL tree's operations include mechanisms to ensure that the tree always exhibits height balance. These mechanisms check the height balance of the tree after each insertion and removal of an element and perform rebalancing operations known as ***rotations*** whenever height balance is lost.

- A rotation is a simple operation that restructures an isolated region of the tree by performing a limited number of pointer updates that result in one node moving "upwards" in the tree and another node moving "downwards." Importantly, this is done in such a way as to preserve the BST property among all nodes in the tree.

- Each rotation has a center and a direction. The center is the node at which the rotation is performed, and we can perform either a ***left rotation*** or a ***right rotation*** around this center node.
    - A left rotation moves nodes in a "counterclockwise" direction, with the center moving downwards and nodes to its right moving upwards.
    - Conversely, a right rotation moves nodes in a "clockwise" direction, with the center moving downwards and nodes to its *left* moving upwards.

- For example, here's what a left rotation would look like:




- In the left rotation depicted above, the node with key 32 is the center of the rotation. This node moves downward in the tree, while its right child, the node with key 64, moves upward.
    - In this simple example, the rotation restores height balance to the tree.

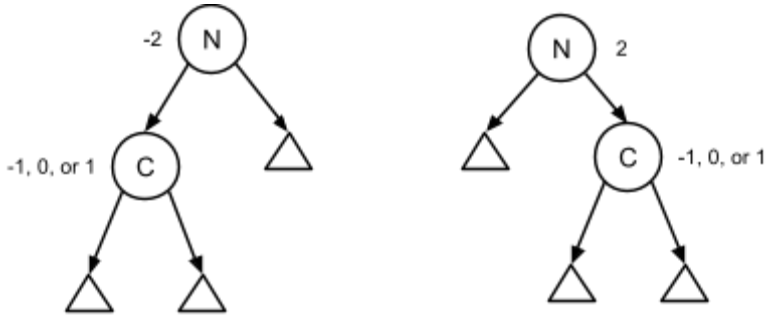
- Sometimes, a ***single rotation*** like the one above, will be enough to restore height balance locally to the tree. Sometimes, however, a ***double rotation*** will be needed. Below, we'll explore the mechanics of these rotations in more detail.

# Deciding how to rotate

- A rotation of some kind (i.e. single or double) will be needed any time an insertion into or removal from an AVL tree leaves the tree (temporarily) with a node whose balance factor is either -2 or 2. In other words, a rotation is needed when height balance is lost at a specific node in the tree. Let's call this node *N*.

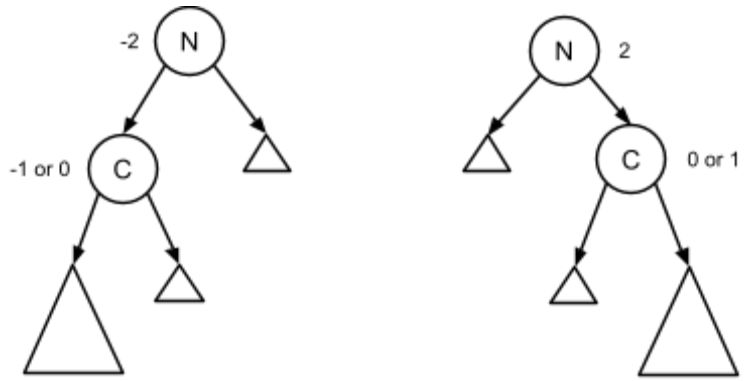
- If *N* has a balance factor of -2, this means *N* is left-heavy. If, on the other hand, *N* has a balance factor of 2, this means *N* is right-heavy. Regardless of the *direction* of *N*'s heaviness, let's refer to the heavier of *N*'s children as *C*.

- The node *C* itself will have a balance factor of -1, 0, or 1, which, respectively, mean that *C* itself is left-heavy, balanced, or right-heavy. In other words, we can have any of the following situations, all of which require some kind of rotation:




N left-heavy N right-heavy

- If ***N* and *C* are heavy in the same direction** (i.e. if the balance factor has the *same sign* at both *N* and *C*) then a single rotation is needed around *N* in the *opposite direction* as *N*'s heaviness. This is also the case when *C* is balanced (i.e. when *C* has balance factor 0). In other words, these are the situations in which a single rotation is needed:

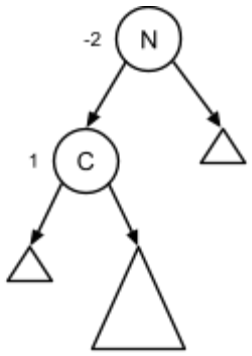



Single right rotation needed

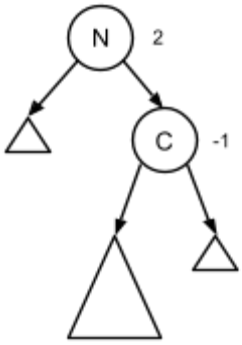
Single left rotation needed

- If, on the other hand, ***N* and *C* are heavy in opposite directions** (i.e. if the balance factor has *opposite signs* at *N* and *C*), then a double rotation is needed.
    - If *N* is left-heavy and *C* is right-heavy (i.e. *N* has a negative balance factor and *C* has a positive balance factor) then we first rotate left around *C* then right around *N*.
    - If *N* is right-heavy and *C* is left-heavy (i.e. *N* has a positive balance factor and *C* has a negative balance factor) then we first rotate right around *C* then left around *N*.

- In other words, these are the situations in which a double rotation is needed:




Double rotation needed: right around C, left around N




Double rotation needed: left around C, right around N

- We can summarize this in the following table:

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="2"><code>balanceFactor(N)</code></td></tr>
<tr><td>-2 (left-heavy)</td><td>2 (right-heavy)</td></tr>
<tr><td rowspan="3"><code>balanceFactor(C)</code></td><td>-1<br>(left-heavy)</td><td rowspan="2">Left-left imbalance<br>Single rotation:<br>right around N</td><td>Right-left imbalance<br>Double rotation:<br>1. right around C<br>2. left around N</td></tr>
<tr><td>0</td><td rowspan="2">Right-right imbalance<br>Single rotation:<br>left around N</td></tr>
<tr><td>1<br>(right-heavy)</td><td>Left-right imbalance<br>Double rotation:<br>1. left around C<br>2. right around N</td></tr>
</table>

- Now that we know what kinds of rotations we'll need to do in various situations, let's explore the mechanics of how those rotations are actually carried out.

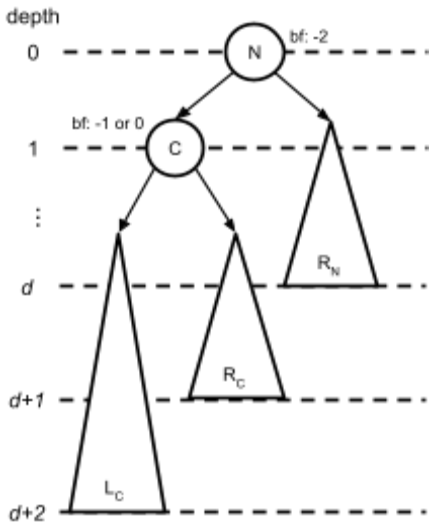
# Single rotations

- 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.

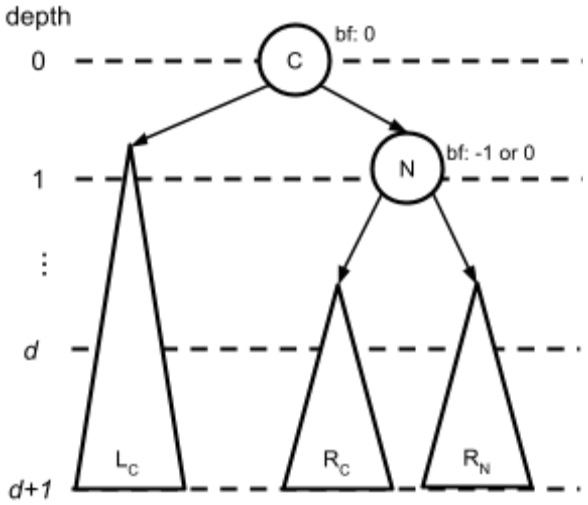
- Again, a single right rotation is needed to restore height balance when there's a left-left imbalance at a node *N* (i.e. *N*'s balance factor is -2 and the balance factor of *N*'s left child *C* is -1 or 0). This could occur after an element is inserted into *C*'s left subtree, causing it to grow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



- Note here that the exact heights of the subtrees $L_C$, $R_C$, and $R_N$ 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 $P_N$, then *C* will replace *N* as $P_N$'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:




- 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 right rotation, the operation above would be mirrored.

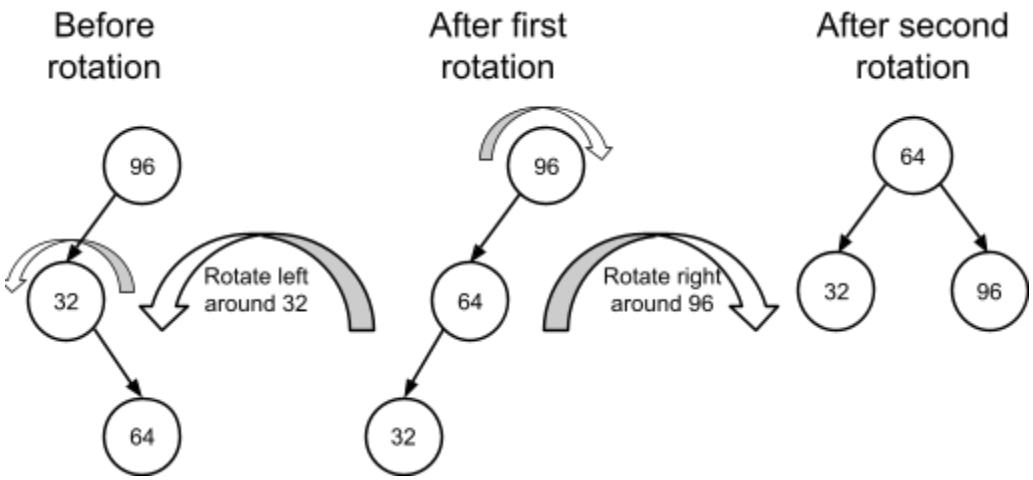
# Double rotations

- 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 96 loses height balance and is left-heavy, and its left child, the node with key 32, is right-heavy:

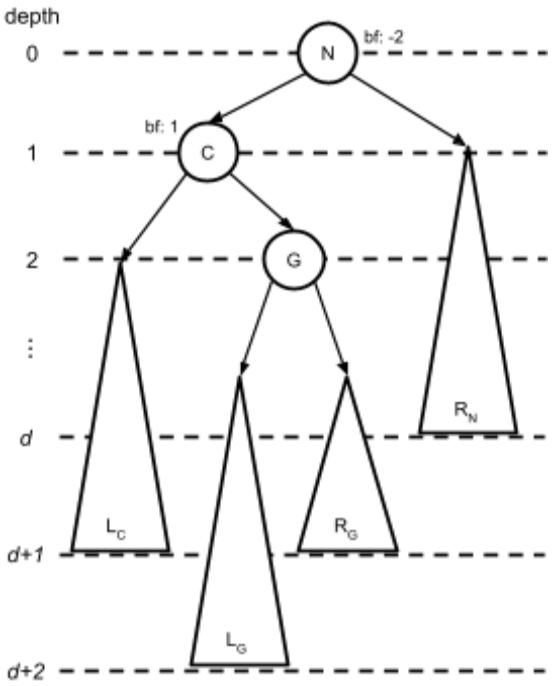


- As the illustration above might make clear, the first rotation in a double rotation is simply intended to align imbalances on the same side, i.e. to create a left-left imbalance or a right-right imbalance. As we saw above, once imbalances are aligned in such a way, a single rotation is sufficient to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

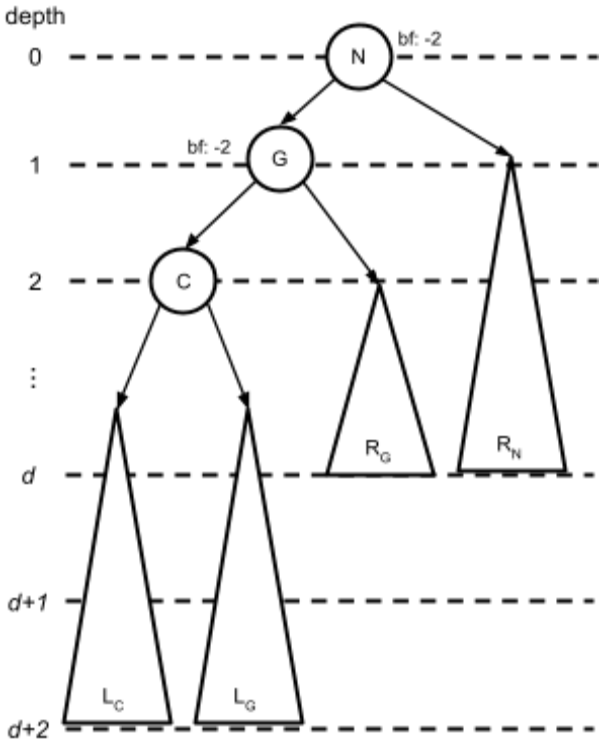
- Let's walk through a more complete visual example of what a double rotation looks like. We'll again focus only on a double rotation to fix a left-right imbalance (i.e. a left rotation around *C* followed by a right rotation around *N*). As with the single rotation we explored above, the double rotation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



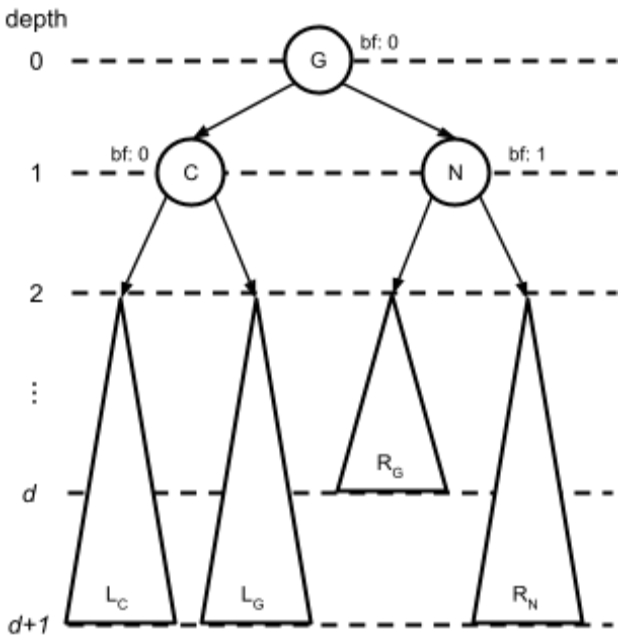
- Again, we'll perform a double rotation to fix this left-right imbalance. The first rotation within the double rotation will be centered around *C* and in the opposite direction of *C*'s imbalance, i.e. a left rotation:




- This first rotation works just like (the mirror of) the single rotation we examined above. Specifically:
    - *G* moves up in the tree to replace *C* as *N*'s left child.
    - *C* moves down in the tree to become *G*'s left child.
    - $L_G$ 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*:




- 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, $P_N$, *G* would replace *N* as the child of $P_N$ 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*, $L_G$ and $R_G$. 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*.

# Using rotations within AVL tree operations

- 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.

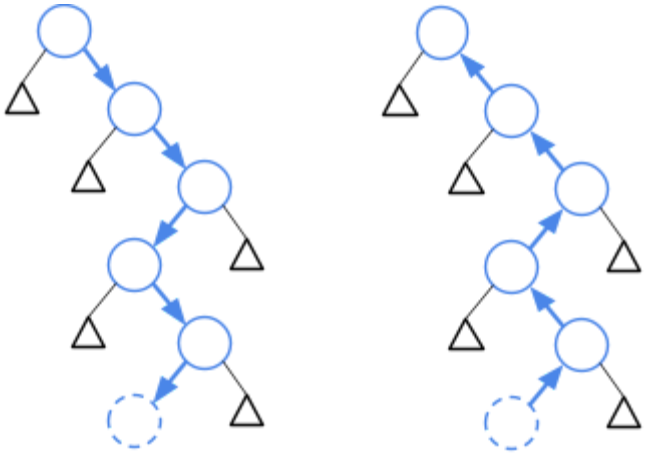
- We also haven't yet seen the mechanics of how an AVL tree recognizes when and where a rotation is needed. For example, how does an AVL tree actually compute a node's balance factor, and how does an AVL tree recognize when a node's balance factor reaches -2 or 2 (i.e. when that node loses height balance)?

- 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:



Path taken downward to location of insertion/removal

Retraced path taken upward to rebalance tree

- 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 C structure to represent each node, it'd look something like this with these modifications:

```
struct avl_node {
     int key;
     void* value;
     int height;
     struct avl_node* left;
     struct avl_node* right;
     struct avl_node* parent;
};
```

- Importantly, like we use NULL to indicate when a node doesn't have a child, we'll also use NULL to indicate when a node doesn't have a parent. Specifically, the root node of the tree will always have a NULL `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.

# Runtime complexity of AVL tree operations

- 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)*.