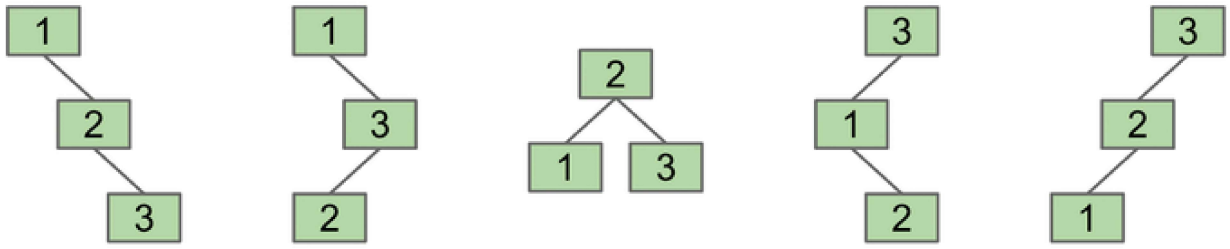
18.1 Rotating Trees

Just like Ferris Wheels.

Wonderfully balanced as they are, B-Trees are really difficult to implement. We need to keep track of the different nodes and the splitting process is pretty complicated. As computer scientists who appreciate clean code and a good challenge, let's find another way to create a balanced tree.

Multiple Structures of BST

For any BST, there are multiple ways to structure it so that you maintain the BST invariants. Earlier, we talked about how **inserting** elements in different orders will result in a different BST. The sequence in which one inserts into a BST will affect its structure. The BSTs below all consist of the elements 1, 2, and 3, yet all have different structures.



> Exercise: For each tree shown above, provide an order of insertion that yields the structure.

Tree Rotation

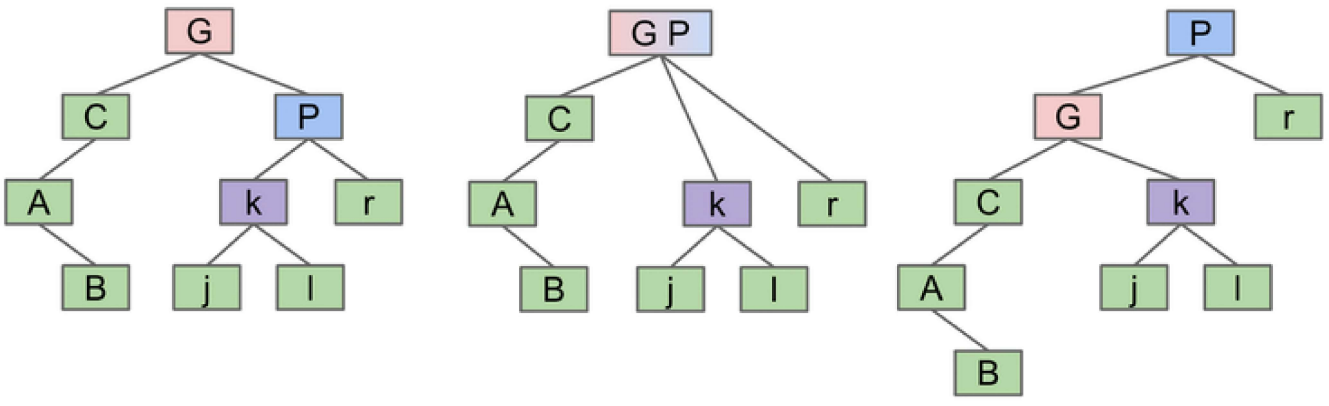
The formal definition of rotation is:

```
rotateLeft(G): Let x be the right child of G. Make G the new left child of x.
```

```
rotateRight(G): Let x be the left child of G. Make G the new right child of x.
```

We will slowly demystify this process in the next few paragraphs.

Below is a graphical description of what happens in a left rotation on the node G:



rotateLeft(G)

The written description of what happened above is this:

- G's right child, P, merges with G, bringing its children along.
- P then passes its left child to G and G goes down to the left to become P's left child.

You can see that the structure of the tree changes as well as its height. We can also rotate on a non-root node. We just disconnect the node from the parent temporarily, rotate the subtree at the node, then reconnect the new root.

Implementation

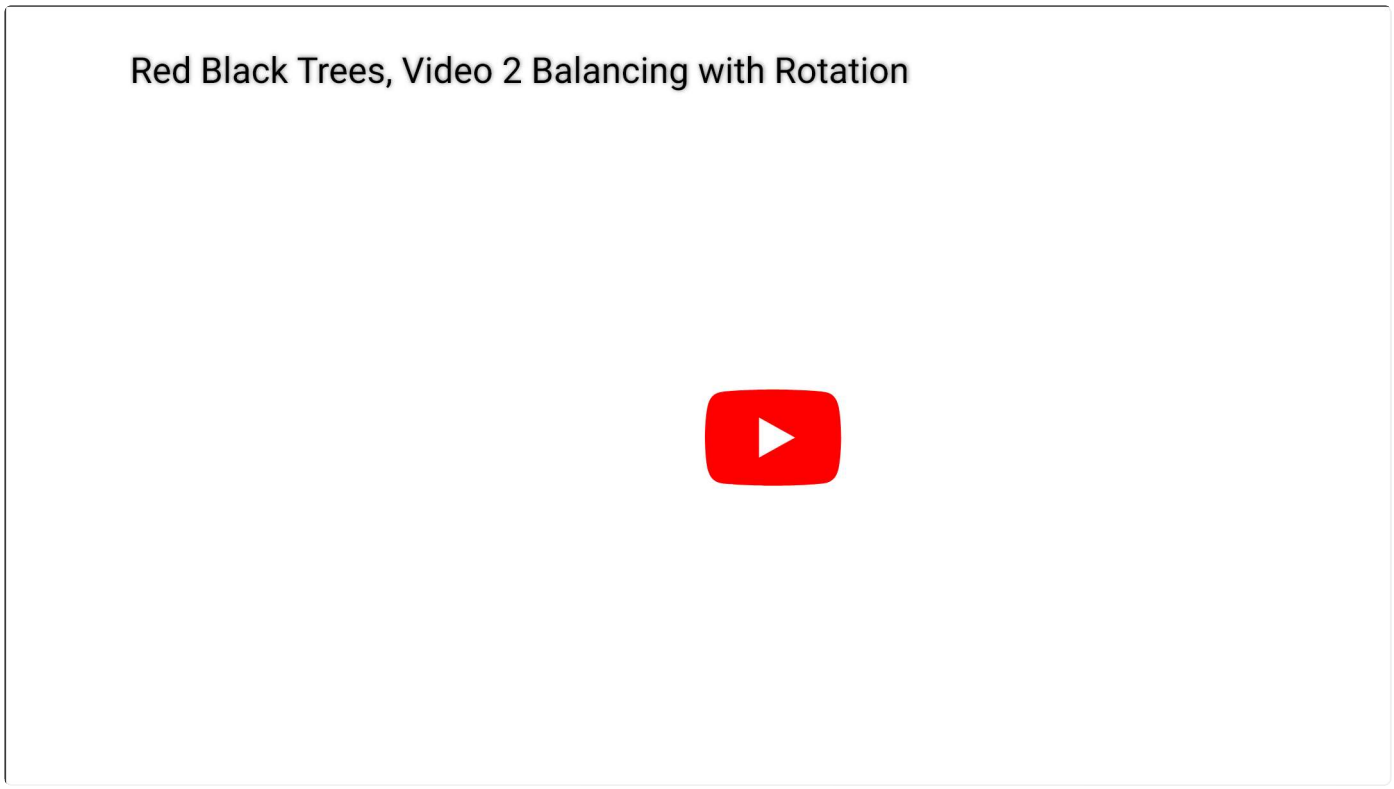
Here are the implementations of `rotateRight` and `rotateLeft:`

```
private Node rotateRight(Node h) {
    // assert (h != null) && isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    return x;
}

// make a right-leaning link lean to the left
private Node rotateLeft(Node h) {
    // assert (h != null) && isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    return x;
}
```

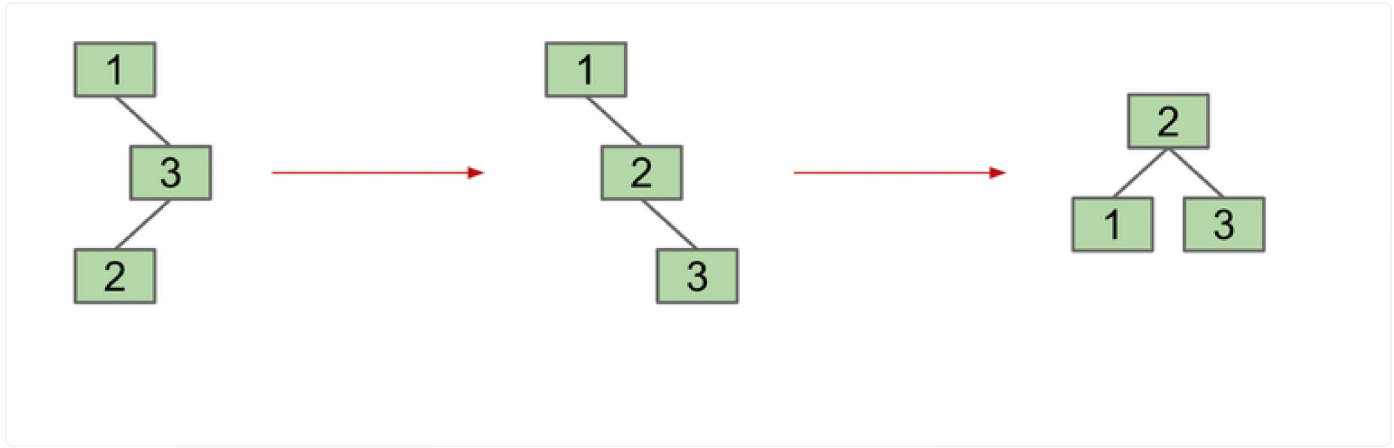
You may be wondering how does the parent node know about which node to point to after we rotate? That's why we are returning x, and other parts of our code will make use of this information to correctly update the parent node's pointer.

Balancing BSTs with Tree Rotation



With rotations, we can actually balance a tree.

Consider the example below:



We can do balance the given BST on the left by calling:

1. `rotateRight(3)`
2. `rotateLeft(1)`

The main observation to make for tree rotation is that it is possible to **shorten** or **lengthen** a tree, while maintaining the search tree's property.

For more examples, see the demo in these slides.

Previous
18. Red Black Trees

Next
18.2 Creating LLRB Trees

Last updated 1 year ago