# CSJ61B Lecture 18

Monday, March 30, 2020

Today we will be discussing 2-3 trees, and 2-3-4 trees, which are a real pain to implement. They suffer from performance problems, which include:

- Maintaining different node types.
- Interconversion of nodes between 2-nodes and 3-nodes
- Walking up the tree to split nodes.

Here's an implementation of 2-3 trees by Kevin Wayne:

```
public void put(Key key, Value val) {
   Node x = root;
   while (x.getTheCorrectChildKey(key) != null) {
      x = x.getTheCorrectChildKey();
      if (x.is4Node()) { x.split(); }
   }
   if (x.is2Node()) { x.make3Node(key, val); }
   if (x.is3Node()) { x.make4Node(key, val); }
}
```

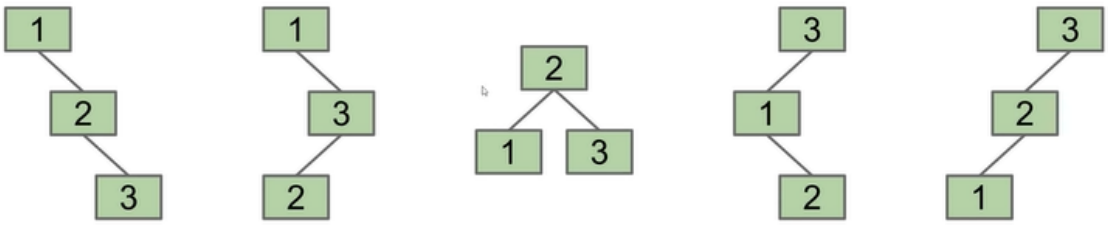
> "Beautiful algorithms are, unfortunately, not always the most useful." - Knuth

We're going to work through a different idea that has a similar underlying concept as 2-3 Trees, but are simpler and faster.

## BST Structure and Tree Rotation

Our new implementation will be based on a concept known as tree rotation. Before we can get to what that is, we must first discuss BSTs.

Suppose we have a BST with numbers 1, 2, 3. There are five valid Binary Search Trees for these 3 numbers, and the order will be based on how they are passed into the algorithm.



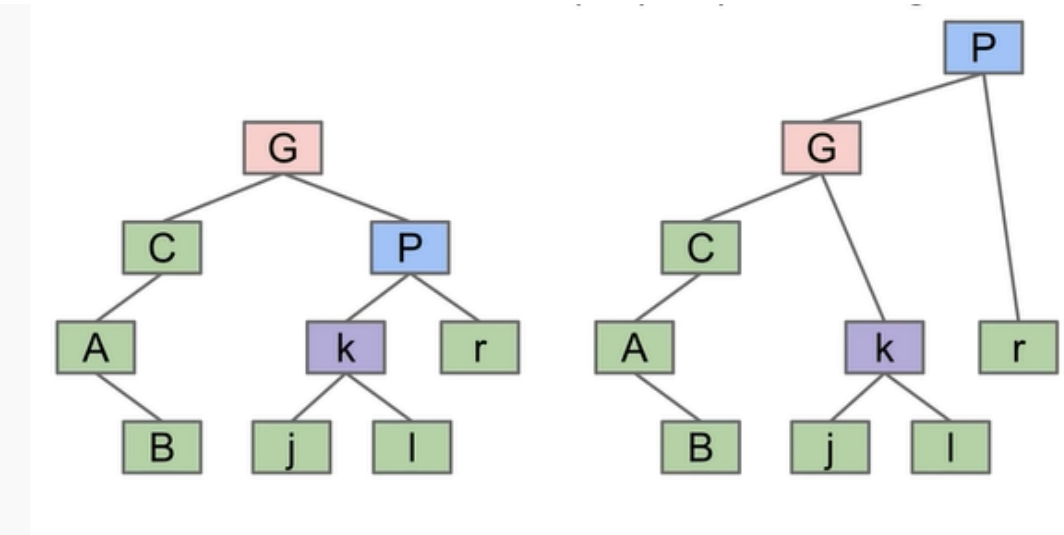

More specifically, the number of valid BSTs is described by a sequence called the Catalan numbers, which pop up a lot in CS.

Given any BST, it is possible to move to a different configuration using "rotation". In general, we can move from any configuration to any other in 2n - 6 rotations.

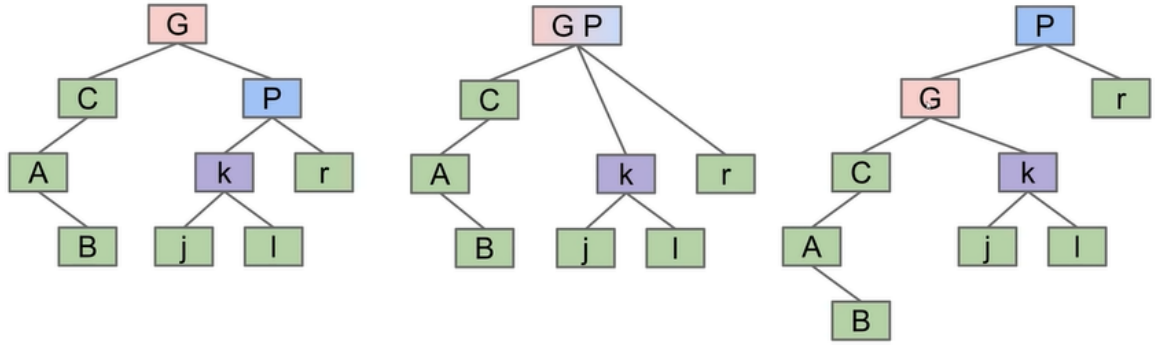
### Tree Rotation

The idea is that we pick a specific node, and rotate it either to the left, or to the right.

`rotateLeft(G)` : Let `x` be the right child of `G`. Make `G` the new left child of `x`. It must preserve the search tree property.




There's an alternate way of thinking about rotation, and that is to imagine we are temporarily merging G and P, then sending G down.

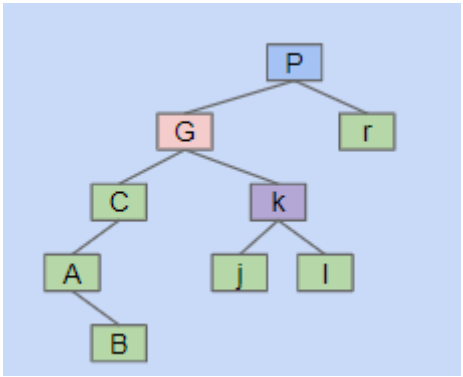



For this example rotateLeft(G) increased height of tree!

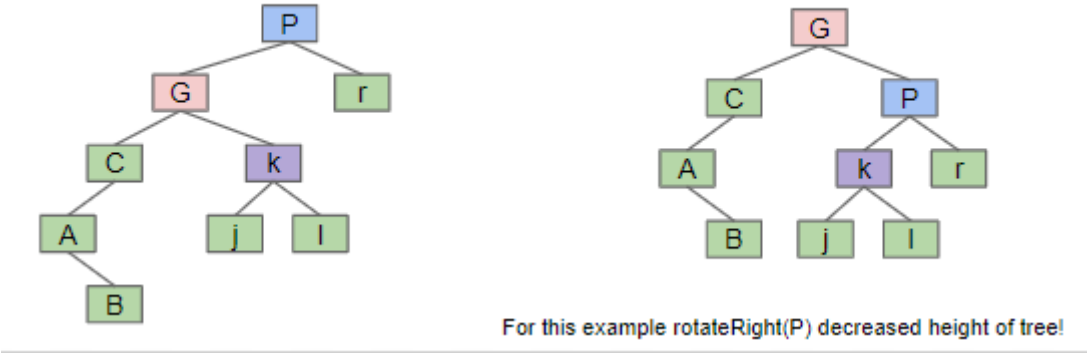
It's important to notice here: there's no change to the semantics of the tree, just its layout.

**Practice**

Try and work through `rotateRight(P)`, letting `x` be the left child of `P`, and making `P` the new right child of `x`.

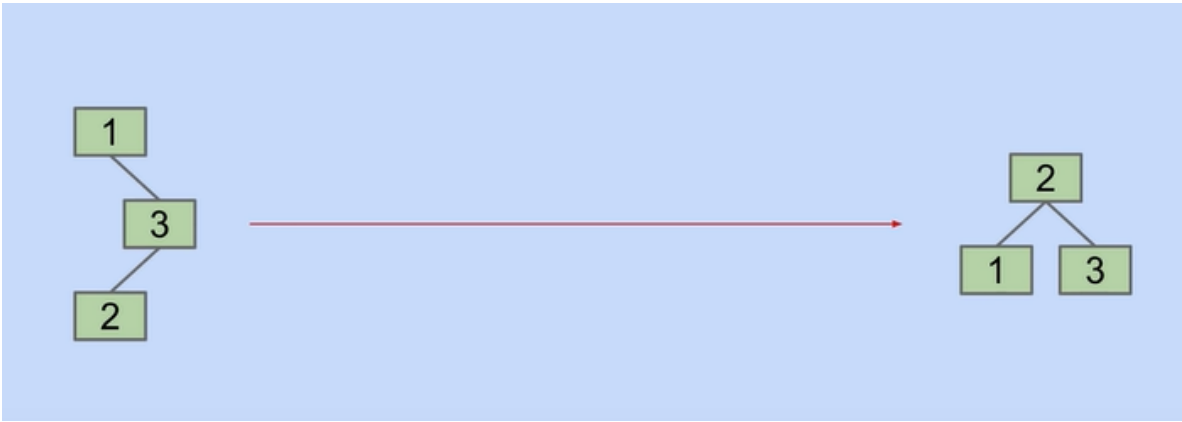



**Solution**



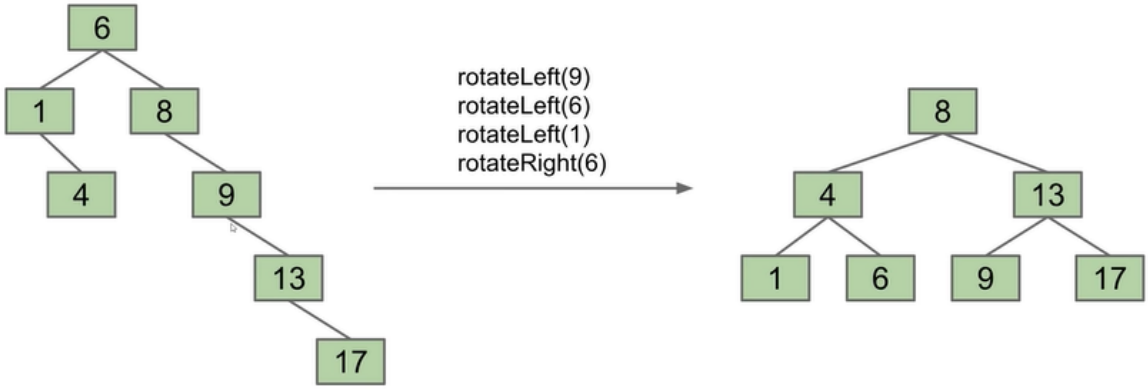

## BST Rotation

Let's work through a problem: give a sequence of rotation operations that balances the tree on the left.




One possible solution is to `rotateRight(3)`, then `rotateLeft(1)`.

Using these, we are able to move from a non-optimal BST to a BST in an optimal state. It is really handy for shortening, lengthening and balancing trees, while preserving the search tree property.




What we don't have at this stage, is some kind of precise algorithm for deciding which algorithms to do. Rotation allows balancing of a BST in O(N) moves, but paying O(N) to occassionally balance a tree is not ideal. Instead of making a tree, letting it get unbalanced, then fixing it, we should do preventative maintenance and keep our tree balanced at all times.

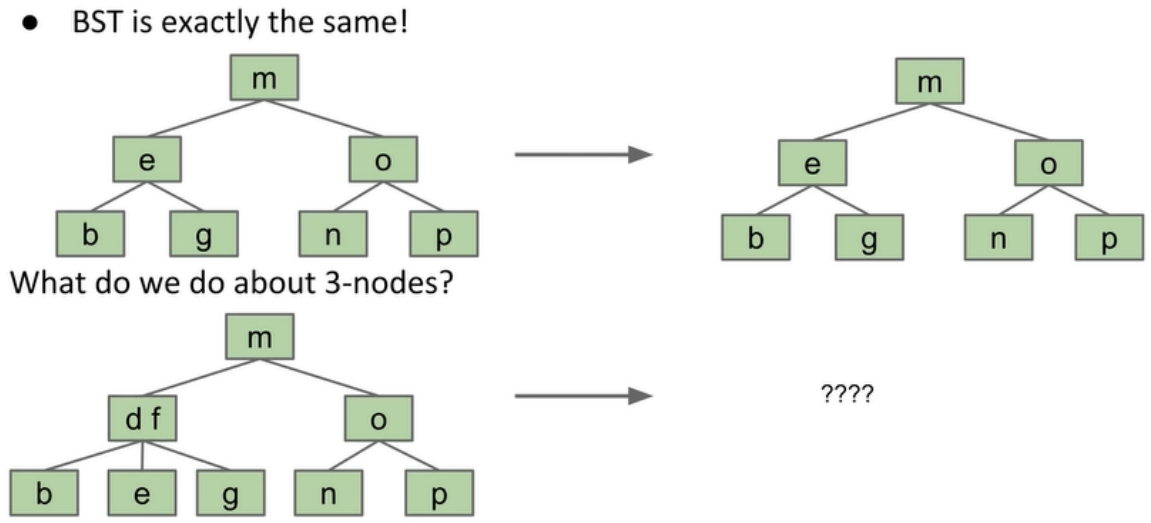
## Red-Black Tree

We now turn our attention to red-black trees, which are widely implemented but often obtuse and difficult to understand.

We've seen two types of BSTs so far: the BST, which can be balanced using rotation, and the BTree (or 2-3 trees, specifically), which are balanced by construction.

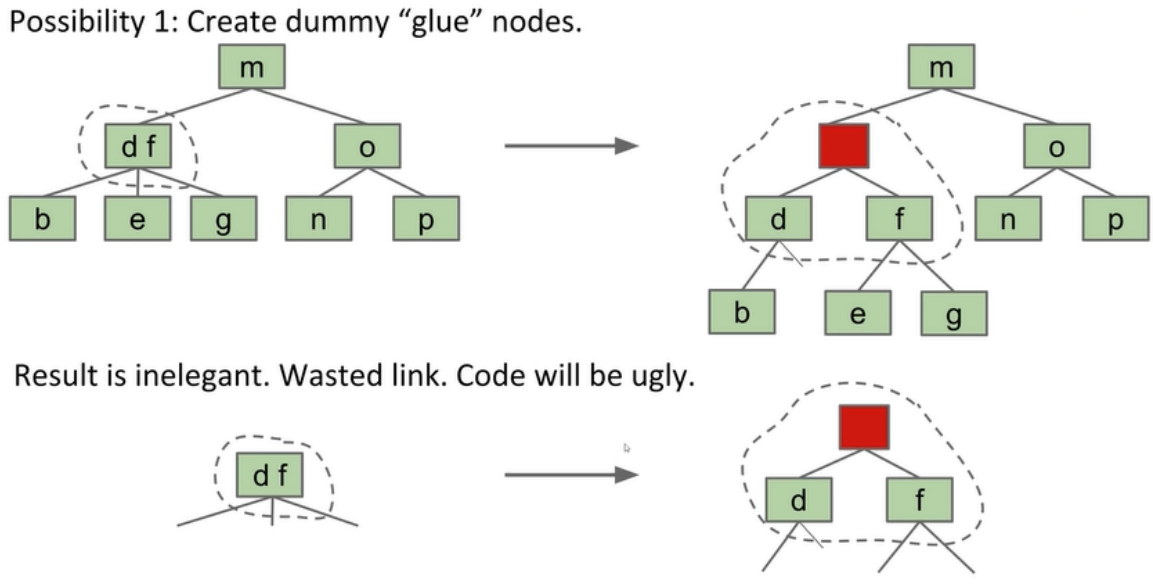
So, we are now going to build a Binary Search Tree that is structurally identical to the 2-3 Tree. Why? Because 2-3 trees are balanced by construction, so will our special BSTs. A 2-3 tree with only 2 nodes is trivial, because it is exactly the same as a BST!

What do we do about 3-nodes?



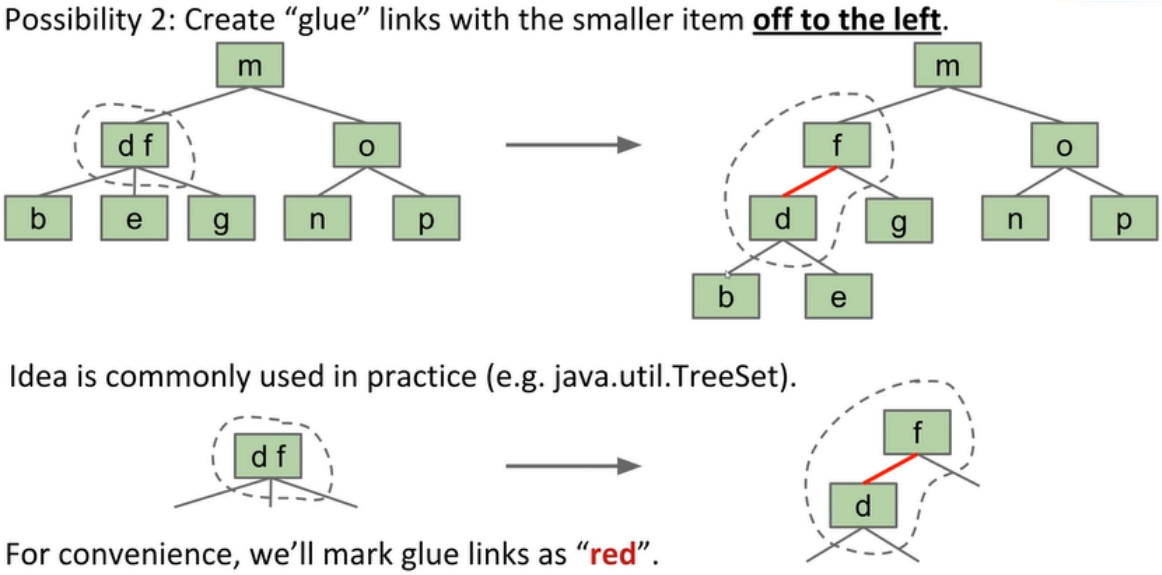

It appears we are, at first stuck, because there's no easy way to represent the 3-node. We can attempt to be clever and do one of several things:

## Create Dummy "Glue" Nodes




The problem with this is that there is some inefficiency here, such as `d` having an unused right link, and not to mention the glue node itself is a little ugly. This isn't, however, what is commonly done.

## Creating "Glue" Links with Offset Items



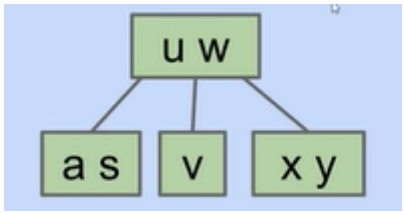

Here, we avoid wasting a link, and the code itself is a little prettier.

# Left-Leaning Red Black Binary Search Tree (LLRB)

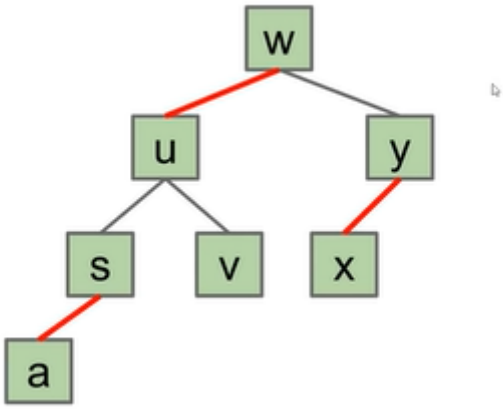
What we've just done here, a BST with left glue links that represent a 2-3 tree, is what is often called the Red-Black Tree. Remember, LLRBs are still normal BSTs, and the red links mean nothing to the computer itself. There is a 1-to-1 correspondence between an LLRB and an equivalent 2-3 tree.

**Practice**

Draw the LLRB corresponding to the 2-3 tree shown below.




**Solution**

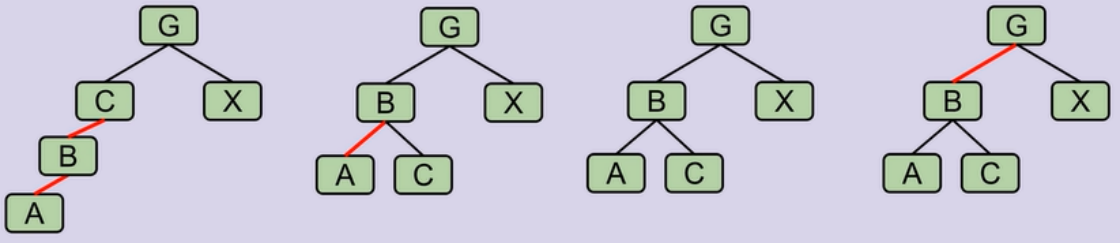



## Properties

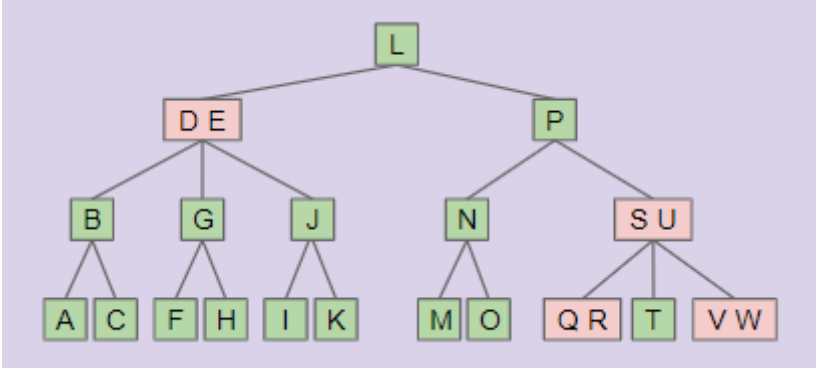
An LLRB can be useful because it can be treated like any BST in the search operation, because it **is** a BST.

**Practice**

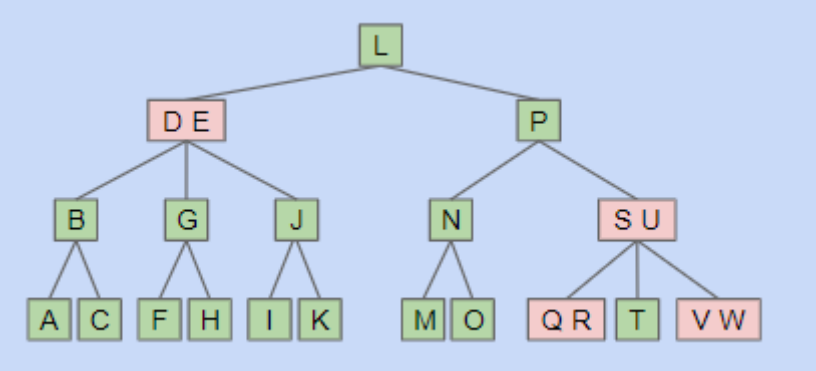
1. How many of these are valid LLRBs?




2. How tall is the corresponding LLRB for the tree below? (3-nodes in pink)

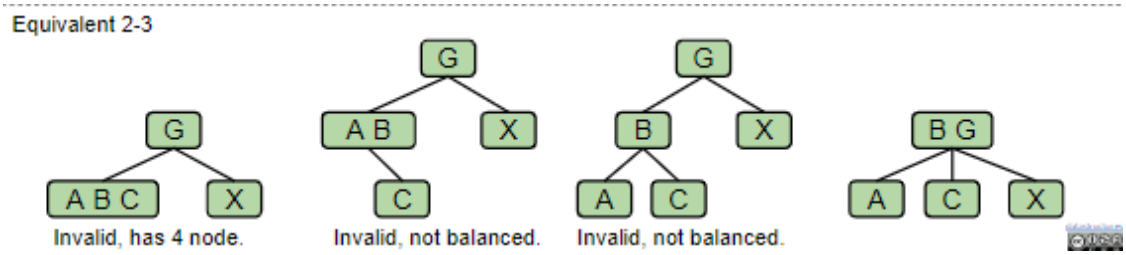



3. Suppose we have a 2-3 tree of height H. What is the maximum height of the corresponding LLRB?

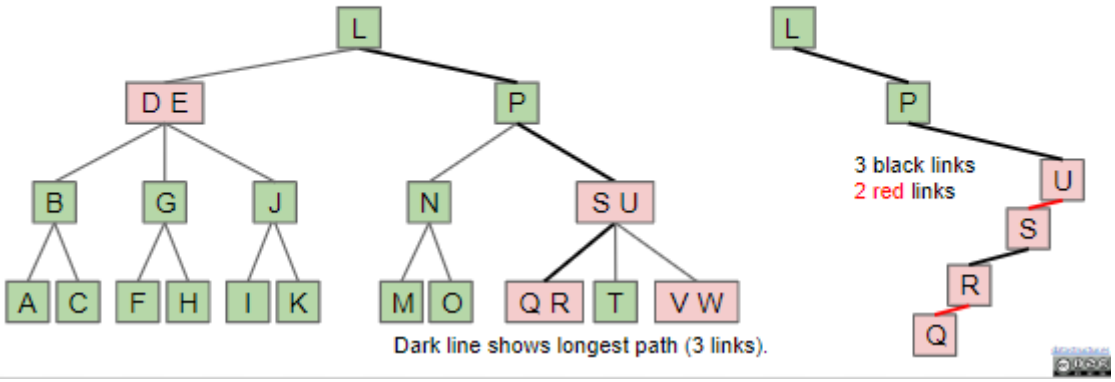



**Solution**

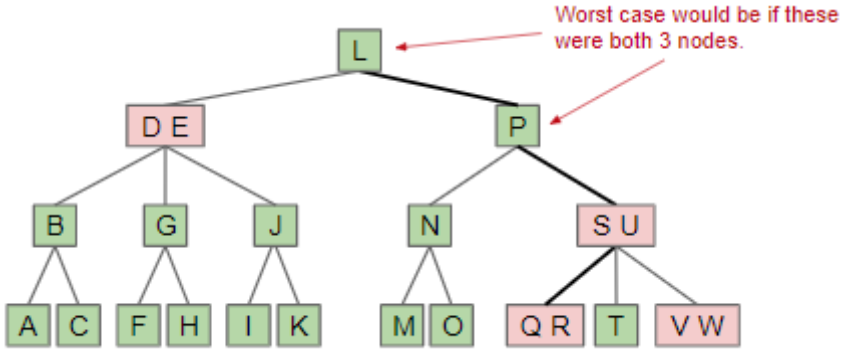
1.



- Each 3-node becomes two nodes in the LLRB.
- Total height is 3 (black) + 2 (red) = 5.
- More generally, an LLRB has no more than ~2x the height of its 2-3 tree.

2. 


3. The total height is H (black) + H + 1 (red) = 2H + 1




Through the above 3 problems, we can make the following conclusions about the properties of an LLRB:

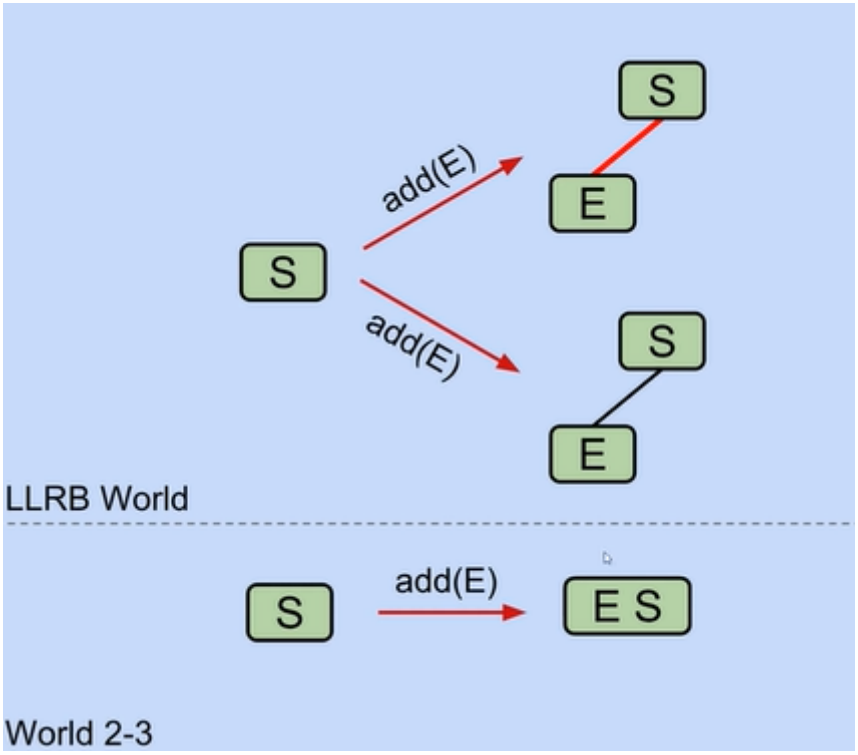
- No node has two red links [otherwise it'd be analogous to a 4 node, which are disallowed in 2-3 trees].
- Every path from root to a null reference has same number of **black links** [because 2-3 trees have the same number of links to every leaf]. LLRBs are therefore balanced. (the only thing that can vary between paths is the number of red links)

## Insertion

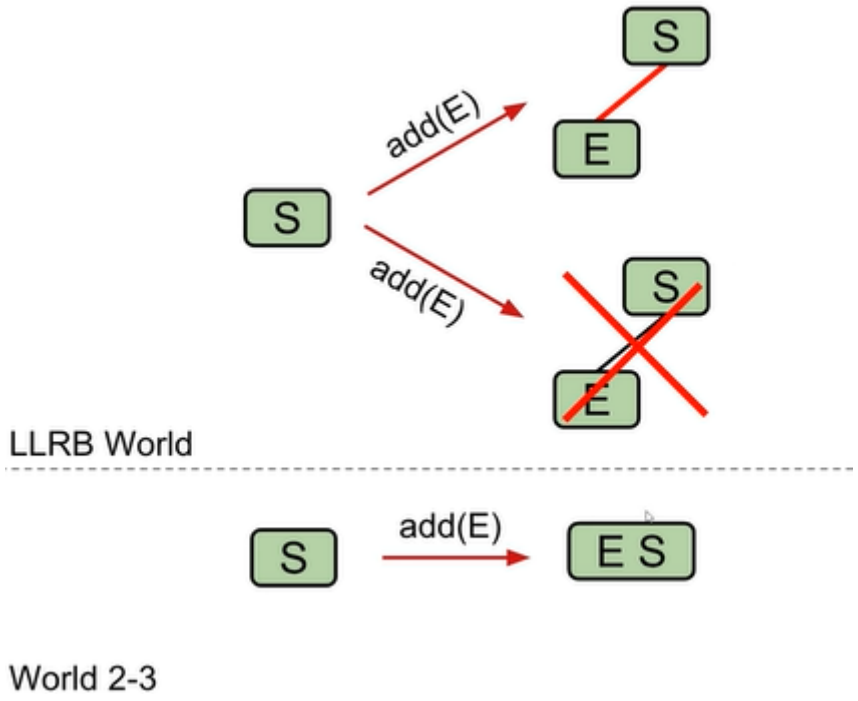
At this point, we know what LLRBs are, and that they're efficient, and now we just need to know how to build them.

There exists a 1-1 mapping between the 2-3 Tree and its corresponding LLRB. Implementing an LLRB is based on maintainin gthis 1-1 correspondence. Doing so requires pretend like we are working a 2-3 tree. Preservation of the correspondence will involve tree rotations.

To build something, we must add things to it. Let's suppose we have an LLRB with only one node, `S`. On the below half of the diagram you will see the LLRB version, while the above half shows us the 2-3 Tree version.

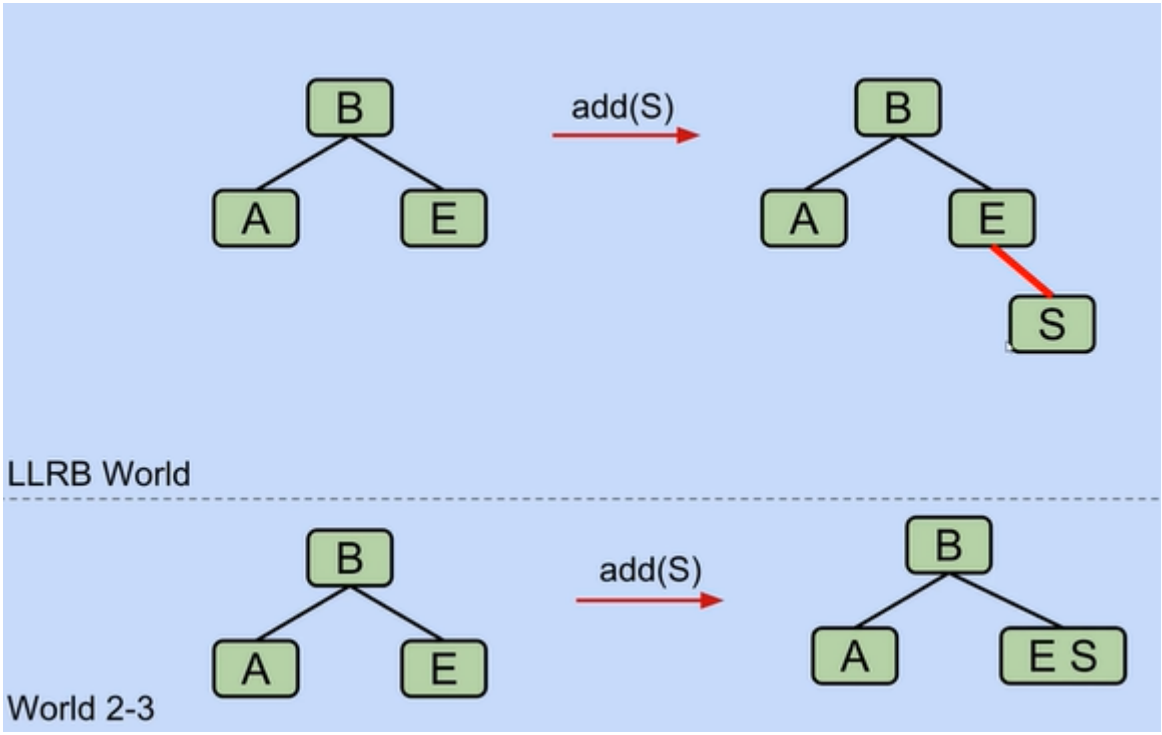



To add to it,

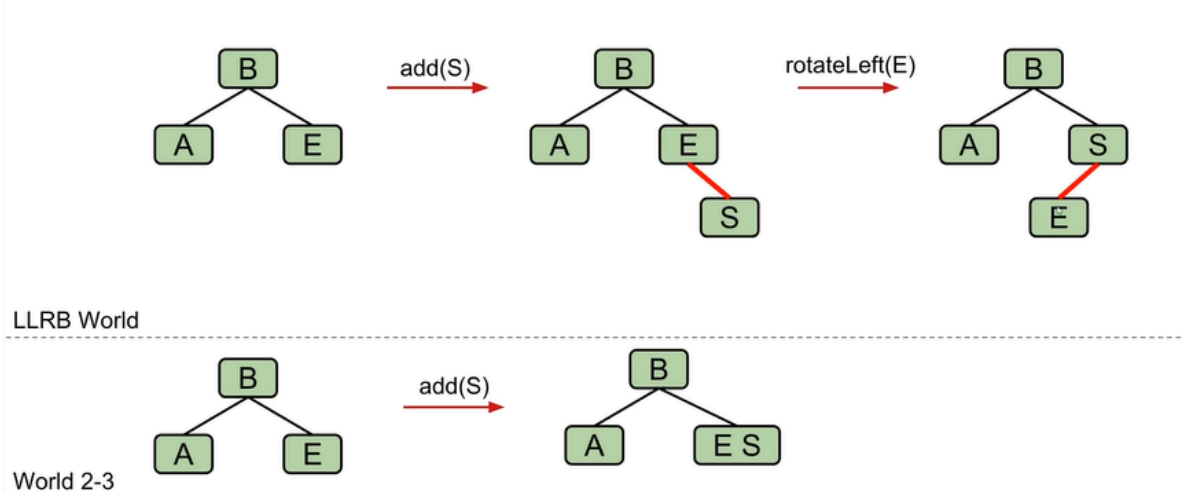



**Practice**

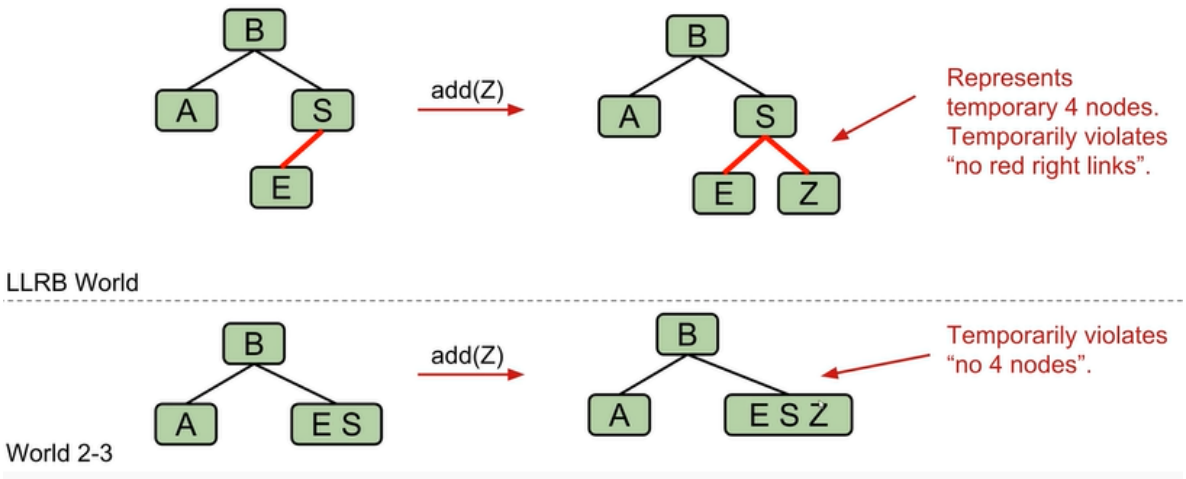
Suppose we have leaf E, and insert S with a red lin. What is the problem below, and what do we do about to maintain its mapping?




**Solution**



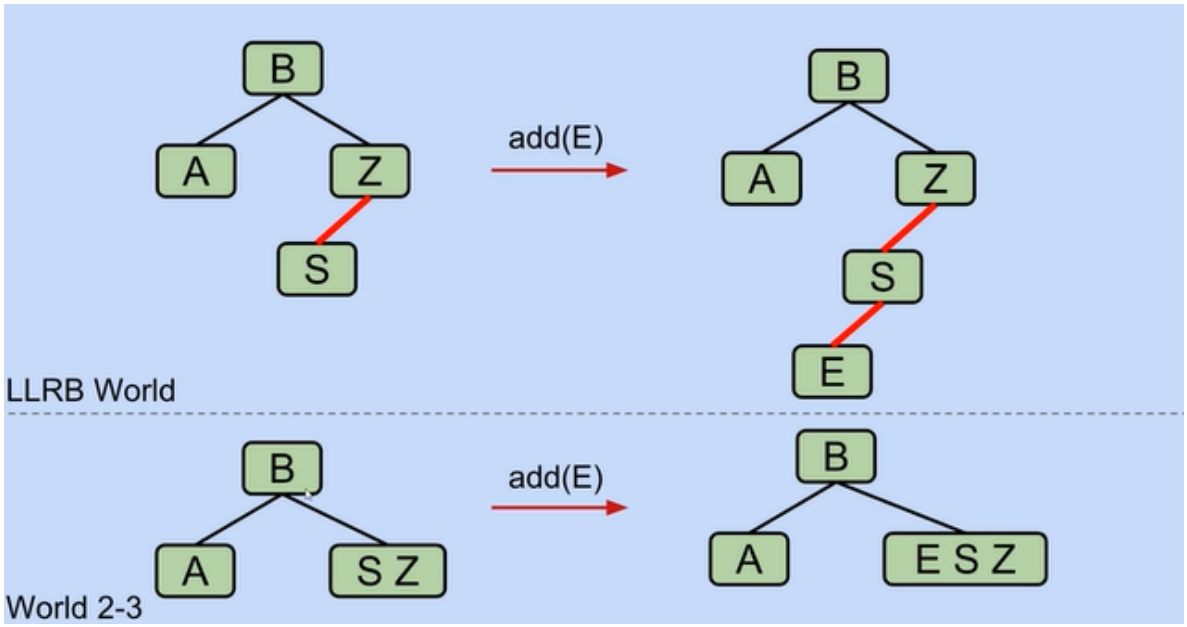

To sort of fix this, we are going to temporarily break our rules and add temporary 4-nodes, represented as BST nodes with two red links. This temporary violation is fine because it will not last long.



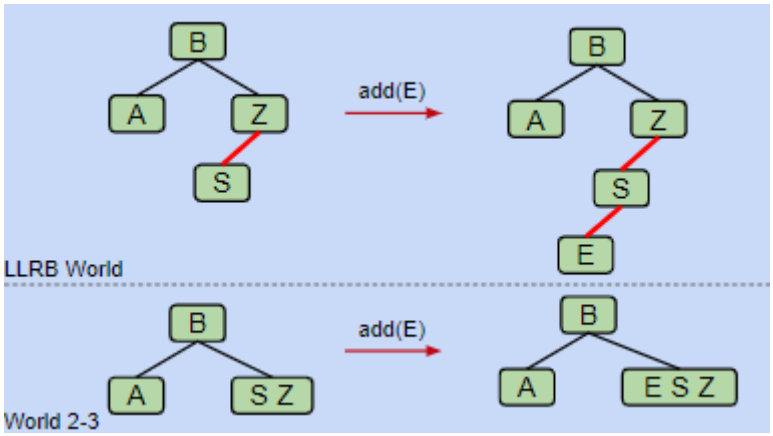

What does this fix?

**Practice**

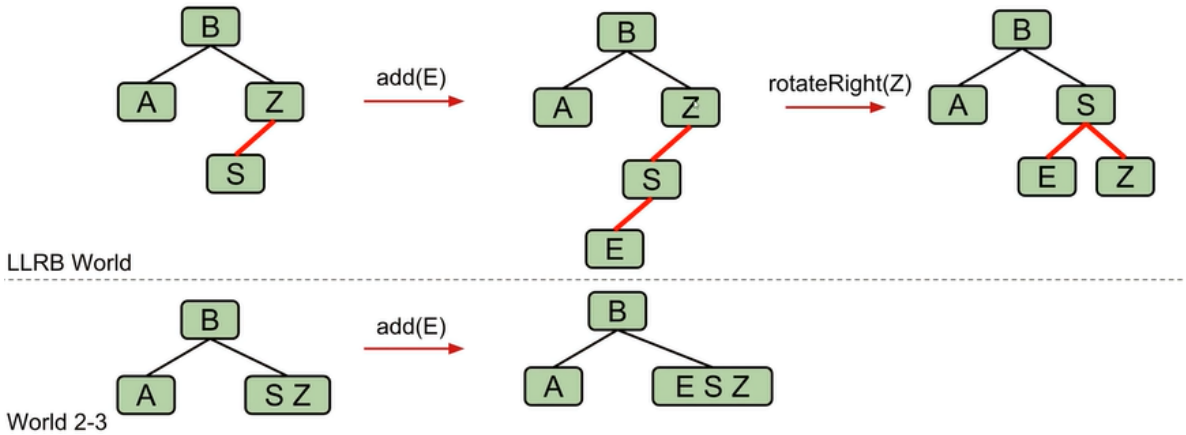
1. Suppose we have the LLRB below and insert E. We end up with the wrong representation for our temporary 4 node. What should we do so that the temporary 4 node has 2 red children (one left, one right) as expected?



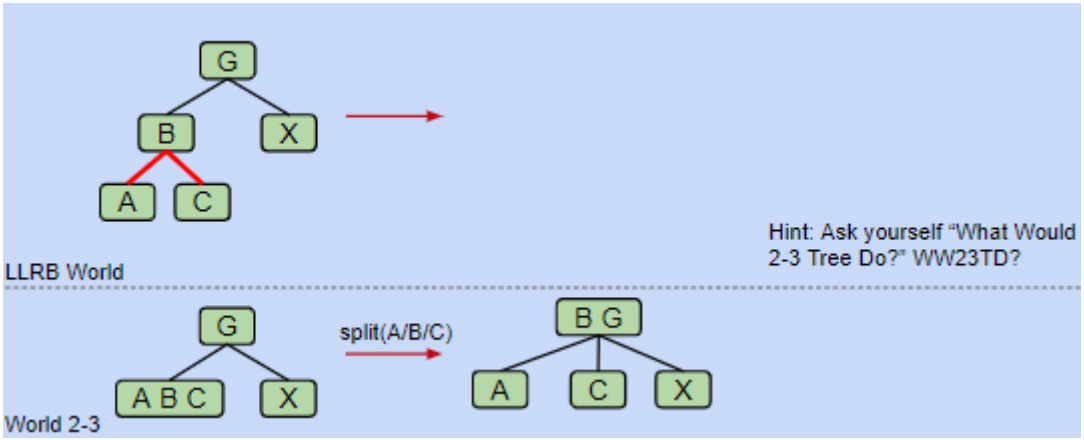

2. Suppose we have the LLRB below which includes a temporary 4 node. What should we do next?




**Solution**




1. This isn't quite right, because we said earlier we should represent the 4-nodes as having both left and right children. We can fix this with `rotateRight(Z)`.
2.

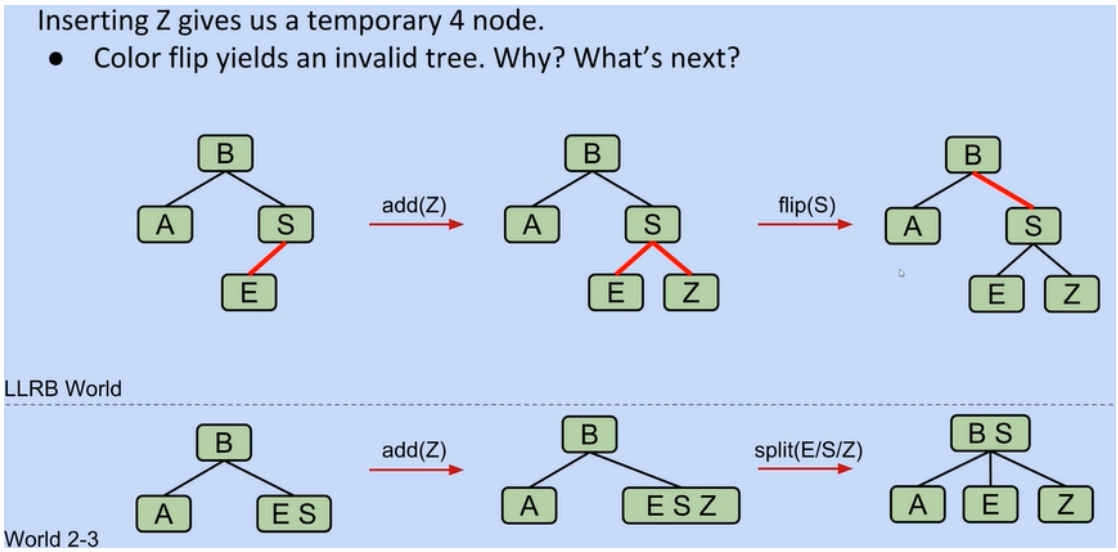



The only thing we have to do here isn't even to rotate the nodes. We just need to `flip(B)`, sich that the colors of all edges touching B are reversed. This doesn't change the BST's structure or shape!

## Summary

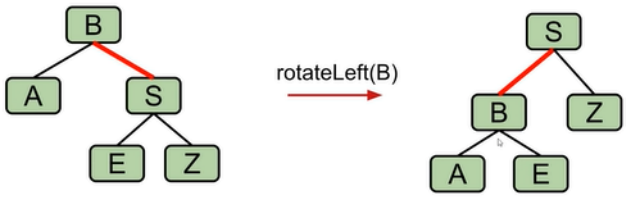
Congratulations, you just invented the red-black BST. Here are some final pointers:

- When inserting: Use a red link.If there is a *right leaning "3-node"*, we have a **Left Leaning Violation.**
  - Rotate left the appropriate node to fix.
- If there are *two consecutive left links*, we have an **Incorrect 4 Node Violation.**
  - Rotate right the appropriate node to fix.
- If there are any *nodes with two red children*, we have a **Temporary 4 Node.**
  - Color flip the node to emulate the split operation.
- One last detail: Cascading operations.It is possible that a rotation or flip operation will cause an additional violation that needs fixing.

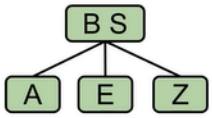
> **Cascading Balance Practice**
>
> 
> 
>
> **Solution**

Inserting Z gives us a temporary 4 node.

- Color flip yields an invalid tree. Why? What's next?
- We have a right leaning 3-node (B-S). We can fix with rotateLeft(b).




LLRB World




World 2-3

# Summary of Search Trees

We have now covered a few different types of search trees that we use to implement sets and maps.

- **Binary search trees** are simple, but they are subject to imbalance.
- **2-3 Trees (B Trees)** are balanced, but painful to implement and relatively slow.
- **LLRBs** insertion is simple to implement (but delete is hard).
  - Works by maintaining mathematical bijection with a 2-3 trees.
- Java's `TreeMap` is a red-black tree (not left leaning).
  - Maintains correspondence with 2-3-4 tree (is not a 1-1 correspondence).
  - Allows glue links on either side (see Red-Black Tree).
  - More complex implementation, but significantly (?) faster.

There are many other types of search trees out there. Other self balancing trees include: AVL trees, splay trees, treaps, etc. There are at least hundreds of different such trees.

And there are other efficient ways to implement sets and maps entirely.

- Other linked structures: Skip lists are linked lists with express lanes.
- Other ideas entirely: Hashing is the most common alternative. We'll discuss this very important idea in our next lecture.