



Celestine Omin

@cyberomin

 Follow



I was just asked to balance a Binary Search Tree by JFK's airport immigration. Welcome to America.

RETWEETS

8,772

LIKES

7,520



8:26 AM - 26 Feb 2017 from [Manhattan, NY](#)

CS61B

Lecture 18: Balanced Search Trees

- Tree Rotation
- Left Leaning Red-Black Trees
- Maintaining Correspondence Through Rotation

The Bad News

2-3 trees (and 2-3-4 trees) are a real pain to implement, and suffer from performance problems. Issues include:

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

fantasy 2-3 code via [Kevin Wayne](#)

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

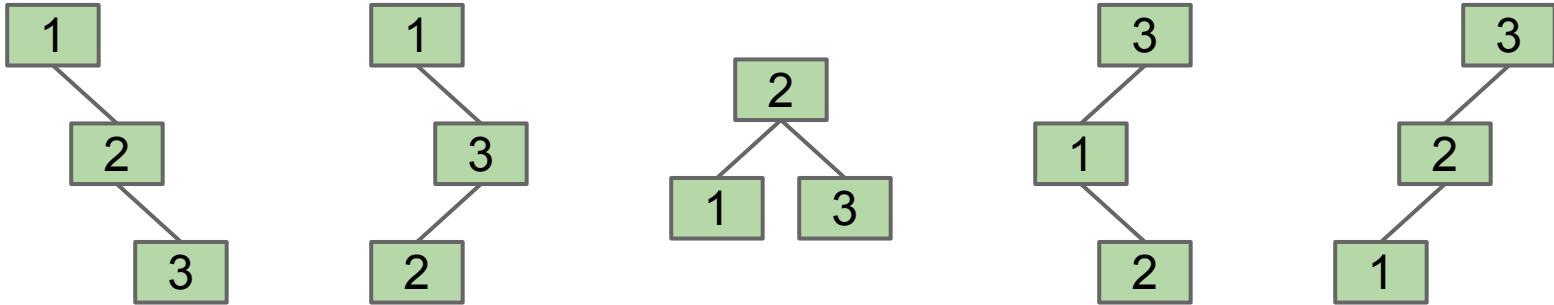
“Beautiful algorithms are, unfortunately, not always the most useful.” - Knuth

BST Structure and Tree Rotation

BSTs

Suppose we have a BST with the numbers 1, 2, 3. Five possible BSTs.

- The specific BST you get is based on the insertion order.
- More generally, for N items, there are [Catalan\(\$N\$ \)](#) different BSTs.



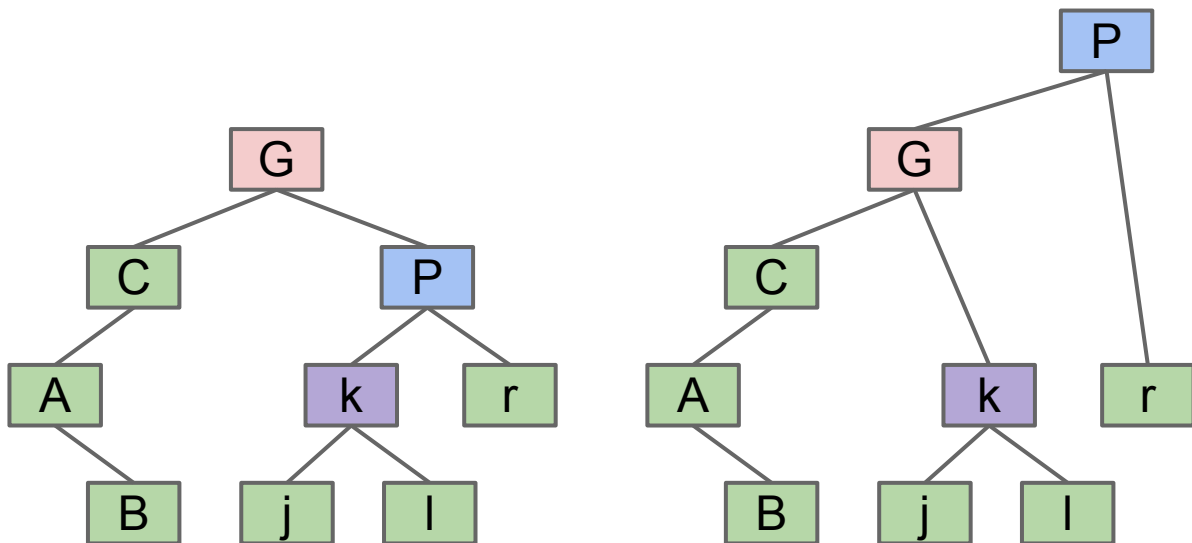
Given any BST, it is possible to move to a different configuration using “rotation”.

- In general, can move from any configuration to any other in $2n - 6$ rotations (see [Rotation Distance, Triangulations, and Hyperbolic Geometry](#) or [Amy Liu](#)).

Tree Rotation Definition

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

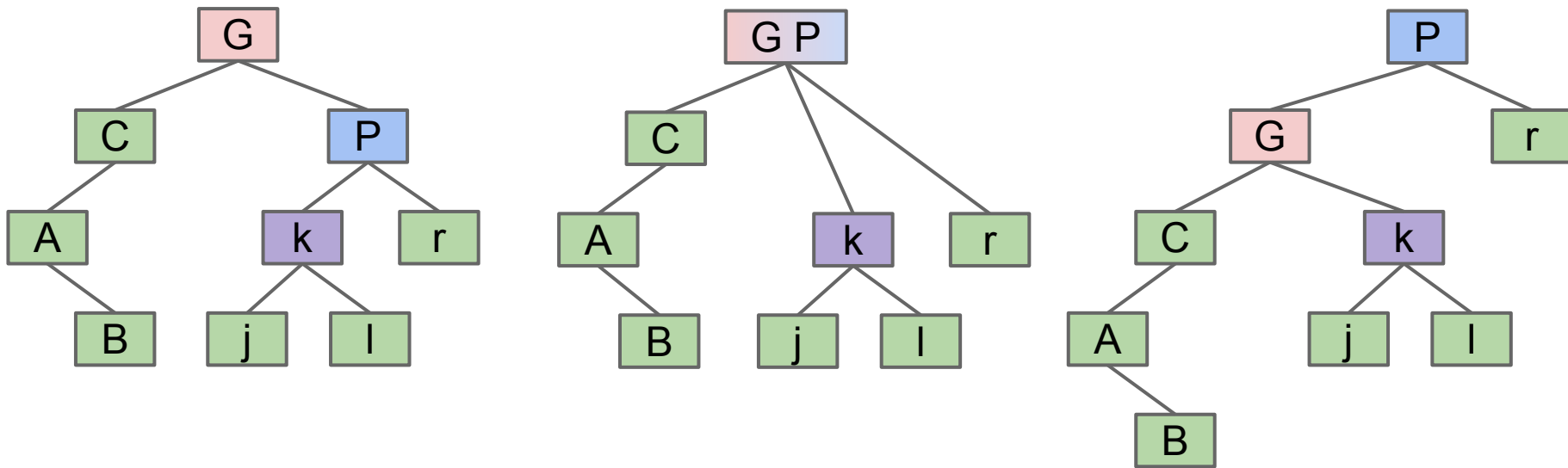
- Preserves search tree property. No change to semantics of tree.



Tree Rotation Definition

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

- Can think of as temporarily merging G and P , then sending G down and **left**.
- Preserves search tree property. No change to semantics of tree.

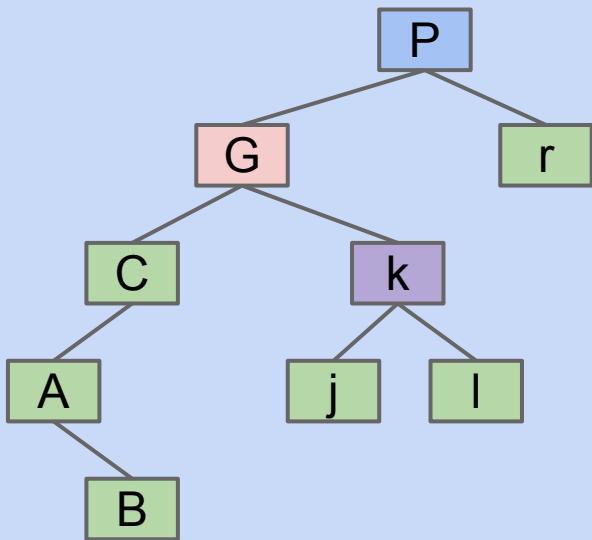


For this example `rotateLeft(G)` increased height of tree!

Your Turn

rotateRight(P): Let x be the left child of P. Make P the **new right child** of x.

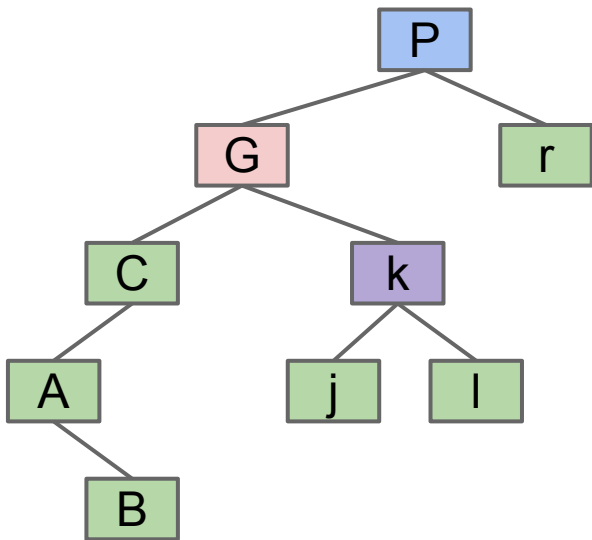
- Can think of as temporarily merging G and P, then sending P down and **right**.



Your Turn

rotateRight(P): Let x be the left child of P. Make P the **new right child** of x.

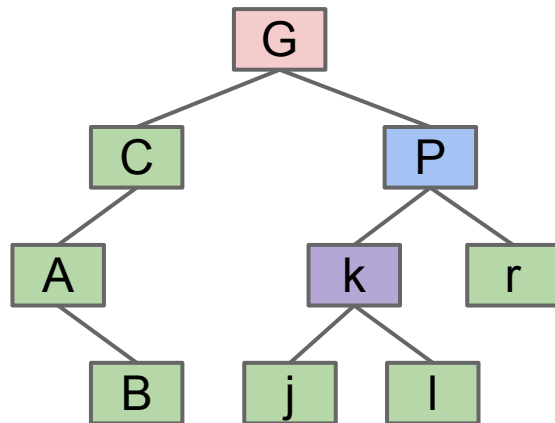
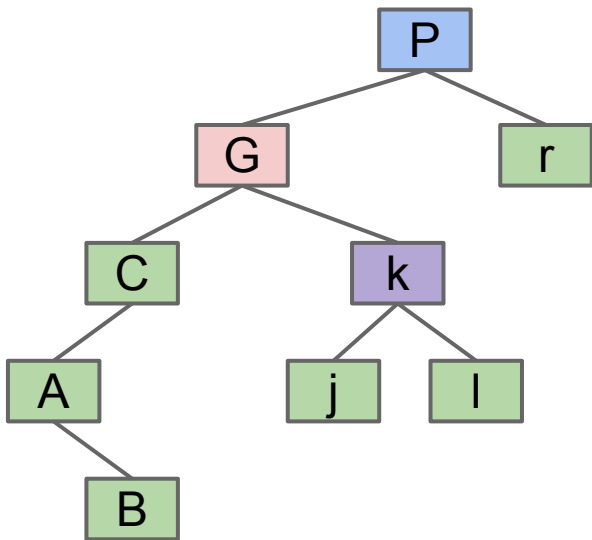
- Can think of as temporarily merging G and P, then sending P down and **right**.



Your Turn

rotateRight(P): Let x be the left child of P. Make P the **new right child** of x.

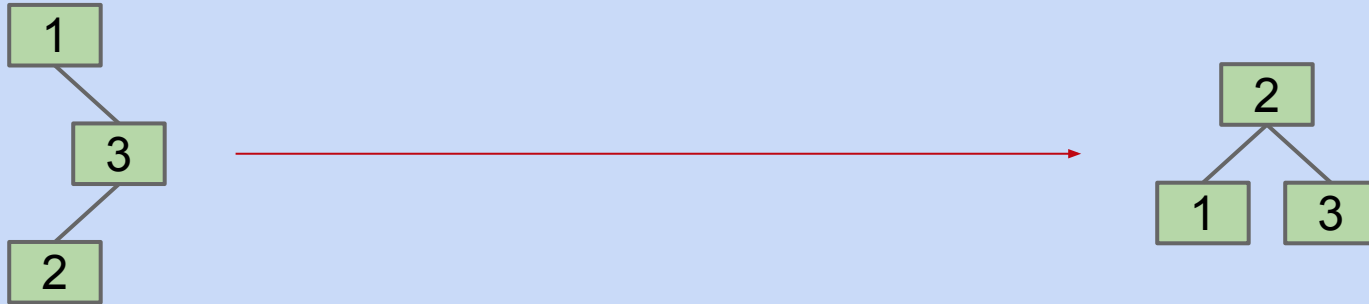
- Can think of as temporarily merging G and P, then sending P down and **right**.
- Note: k was G's right child. Now it is P's left child.



For this example rotateRight(P) decreased height of tree!

BSTs

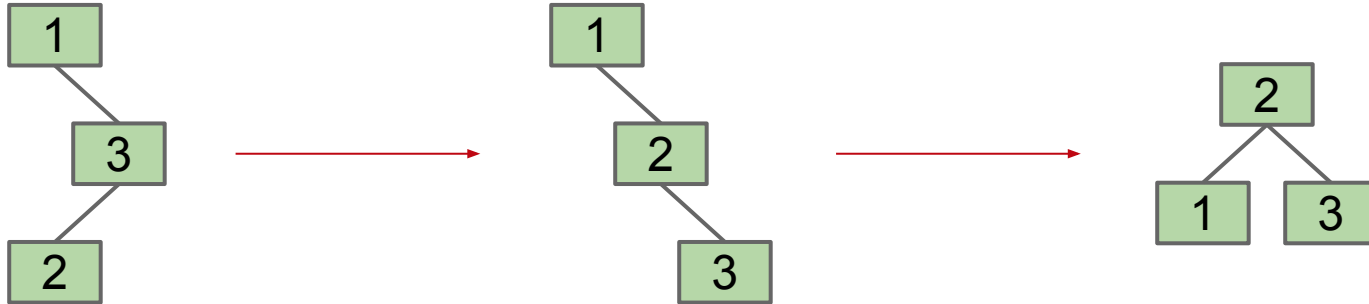
Give a sequence of rotation operations that balances the tree on the left.



BSTs

Give a sequence of rotation operations that balances the tree on the left.

- rotateRight(3)
- rotateLeft(1)

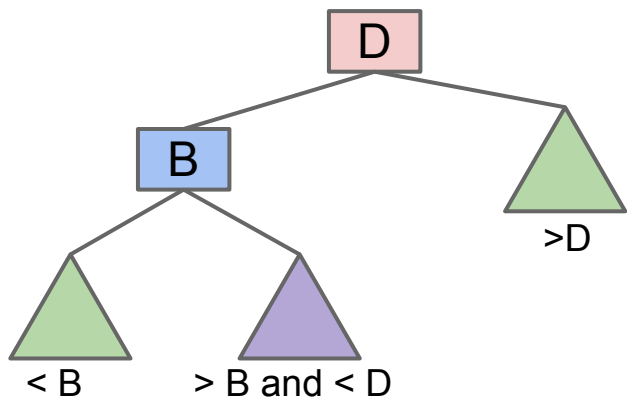


There are other correct answers as well!

Rotation for Balance

Rotation:

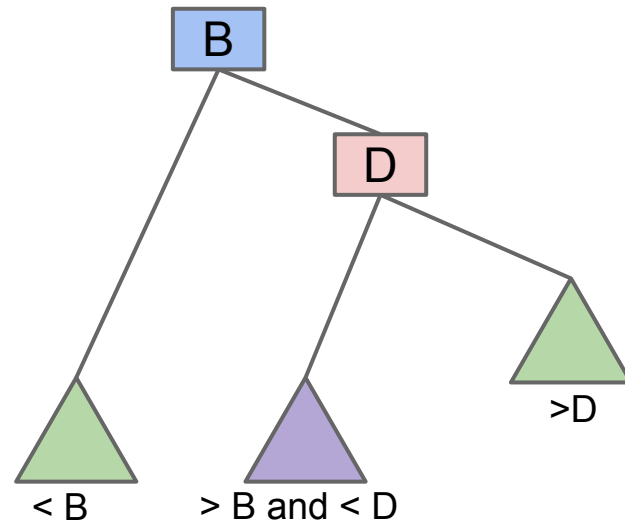
- Can shorten (or lengthen) a tree.
- Preserves search tree property.



rotateRight(D)



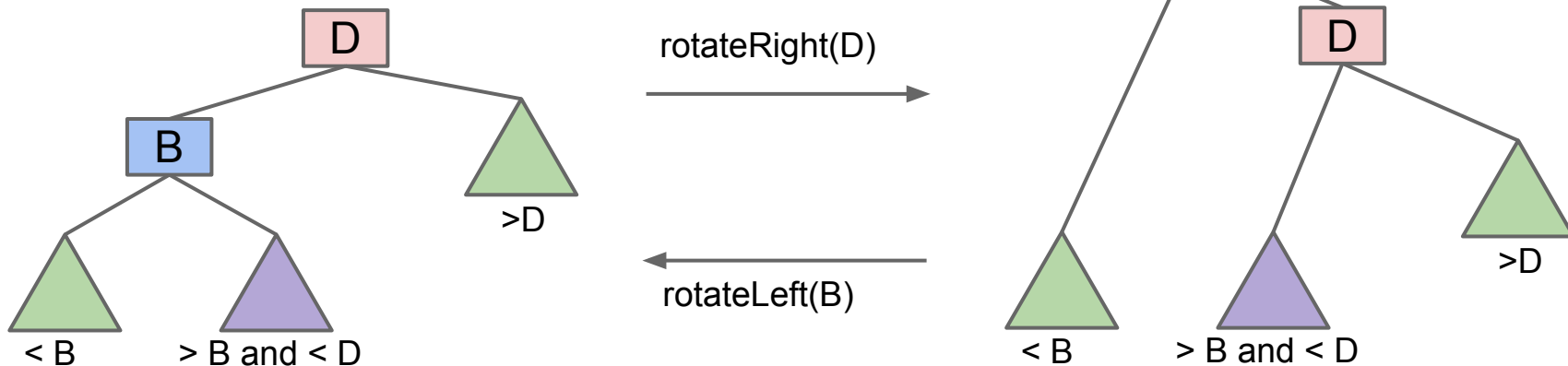
rotateLeft(B)



Rotation for Balance

Rotation:

- Can shorten (or lengthen) a tree.
- Preserves search tree property.



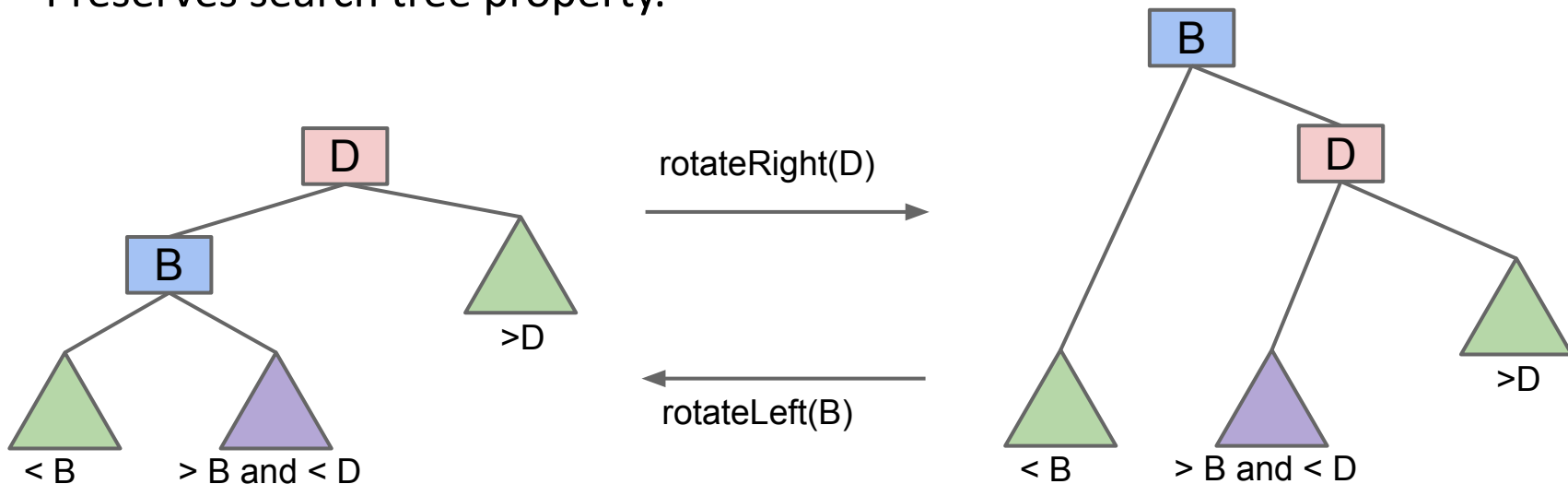
Can use rotation to balance a BST: [Demo](#)

- Rotation allows balancing of a BST in $O(N)$ moves.

Rotation: An Alternate Approach to Balance

Rotation:

- Can shorten (or lengthen) a tree.
- Preserves search tree property.



Paying $O(n)$ to occasionally balance a tree is not ideal. In this lecture, we'll see a better way to achieve balance through rotation. But first...

Red-Black Trees

Search Trees

There are many types of search trees:

- **Binary search trees:** Can balance using rotation, but we have no algorithm for doing so (yet).
- **2-3 trees:** Balanced by construction, i.e. no rotations required.

Let's try something clever, but strange.

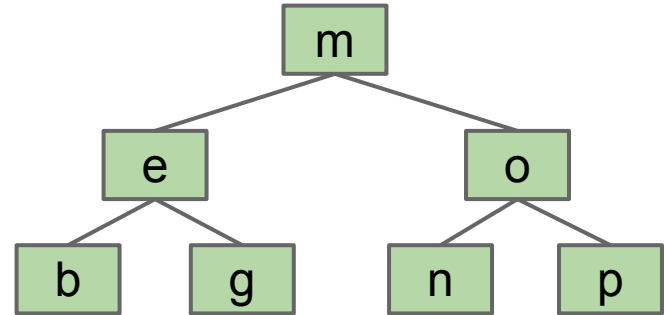
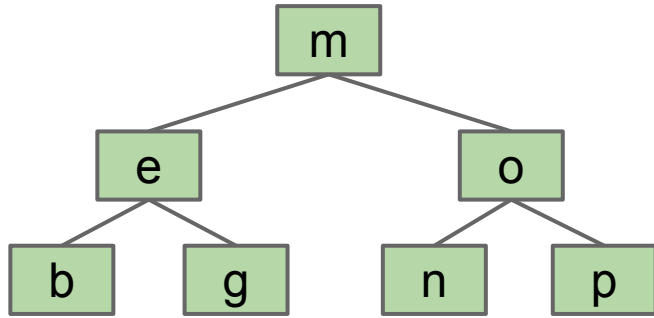
Our goal: Build a BST that is structurally identical to a 2-3 tree.

- Since 2-3 trees are balanced, so will our special BSTs.

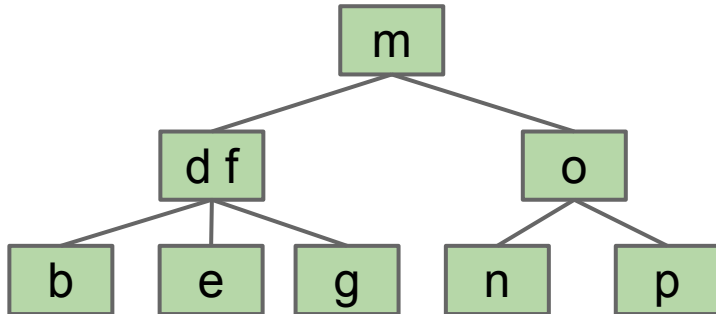
Representing a 2-3 Tree as a BST

A 2-3 tree with only 2-nodes is trivial.

- BST is exactly the same!



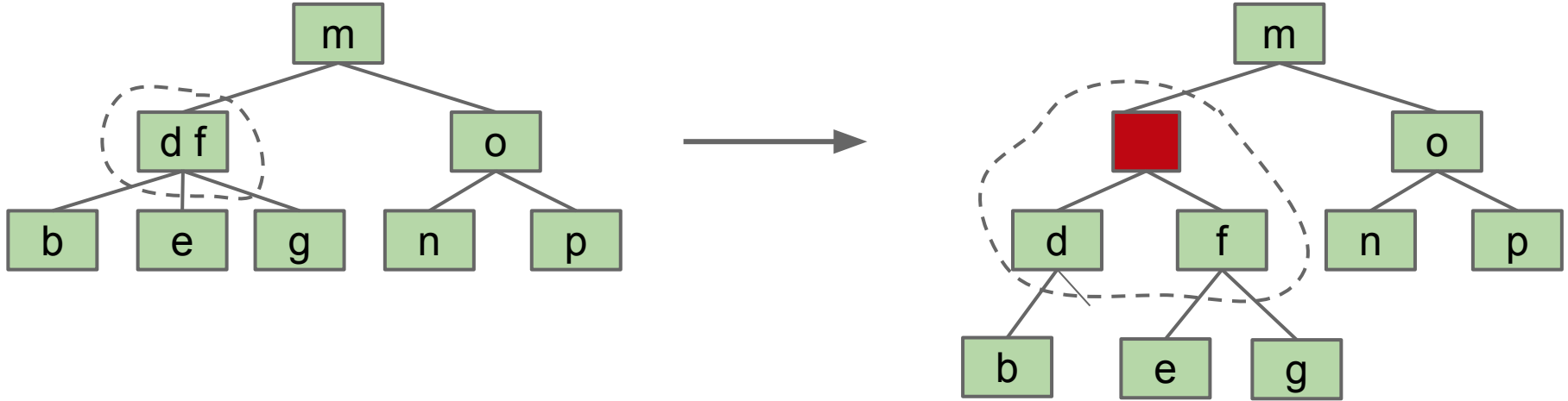
What do we do about 3-nodes?



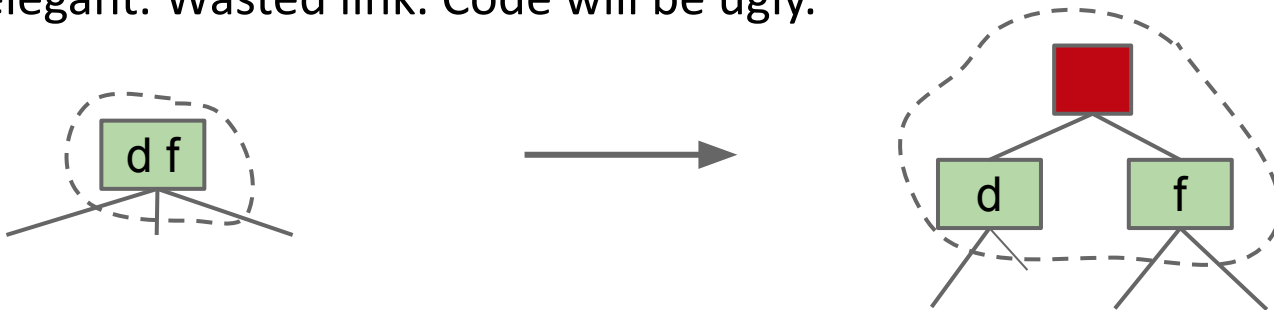
????

Representing a 2-3 Tree as a BST: Dealing with 3-Nodes

Possibility 1: Create dummy “glue” nodes.

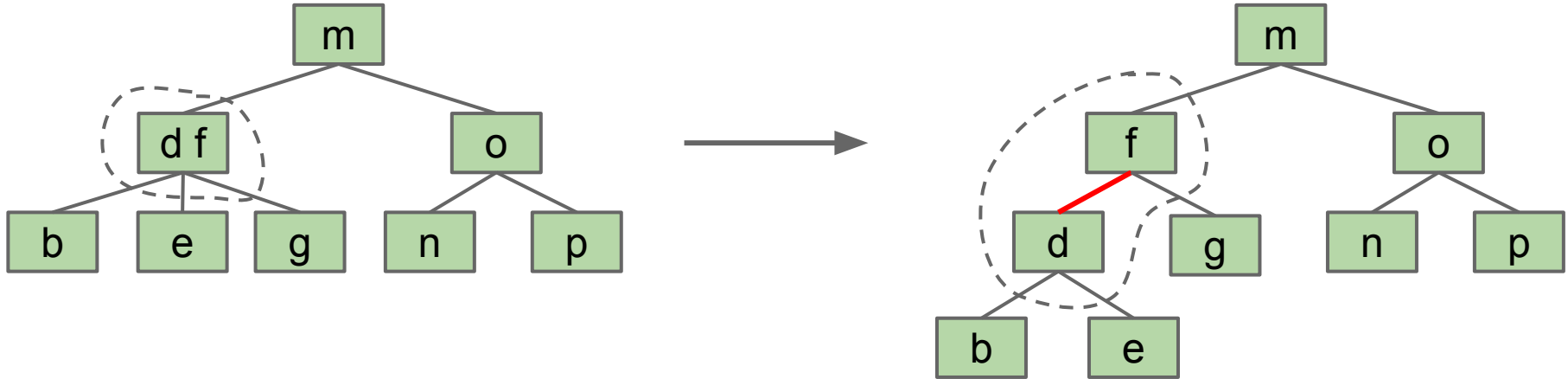


Result is inelegant. Wasted link. Code will be ugly.



Representing a 2-3 Tree as a BST: Dealing with 3-Nodes

Possibility 2: Create “glue” links with the smaller item **off to the left**.



Idea is commonly used in practice (e.g. `java.util.TreeSet`).

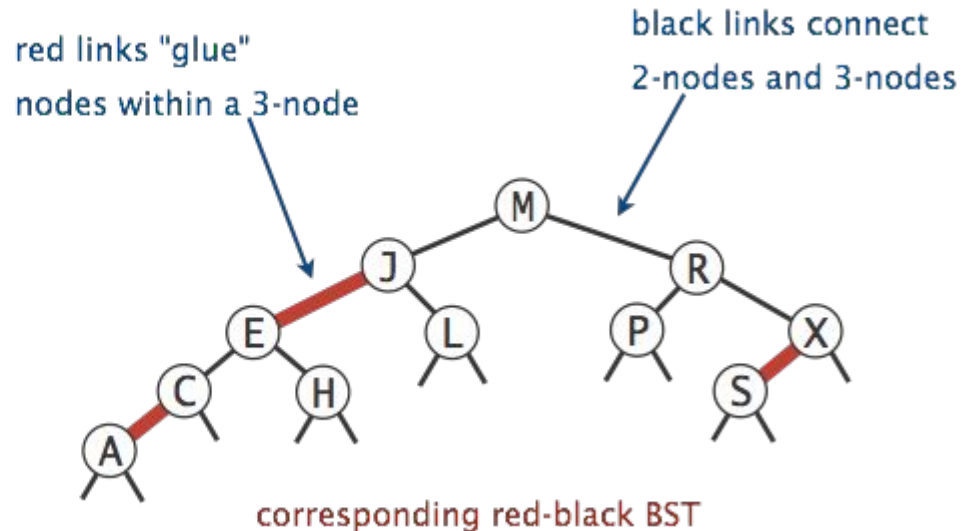
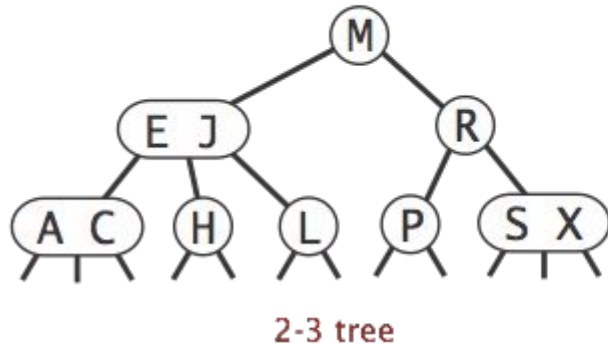


For convenience, we'll mark glue links as “**red**”.

Left-Leaning Red Black Binary Search Tree (LLRB)

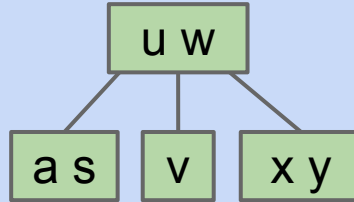
A BST with left glue links that represents a 2-3 tree is often called a “Left Leaning Red Black Binary Search Tree” or LLRB.

- LLRBs are normal BSTs!
- There is a 1-1 correspondence between an LLRB and an equivalent 2-3 tree.
- The red is just a convenient fiction. Red links don’t “do” anything special.



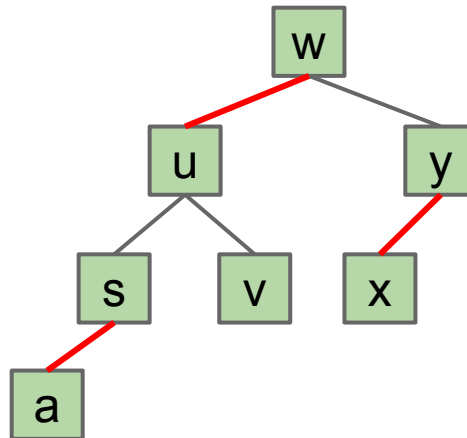
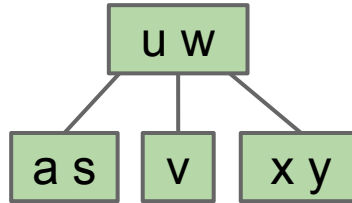
Left-Leaning Red Black Binary Search Tree (LLRB)

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



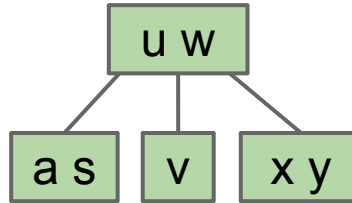
Left-Leaning Red Black Binary Search Tree (LLRB)

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



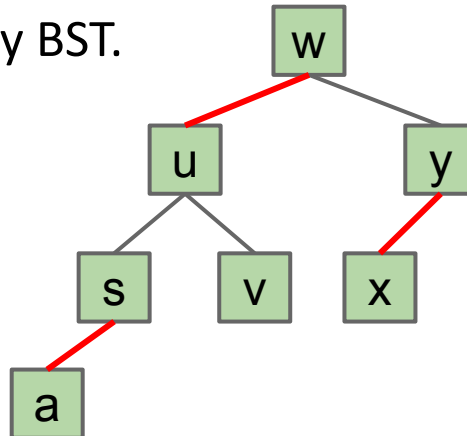
Left-Leaning Red Black Binary Search Tree (LLRB)

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



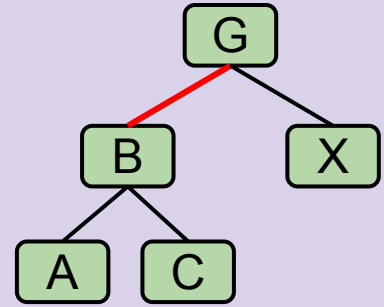
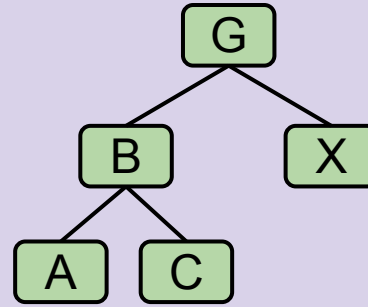
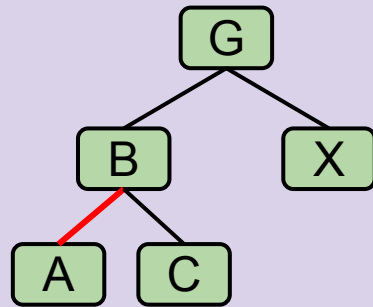
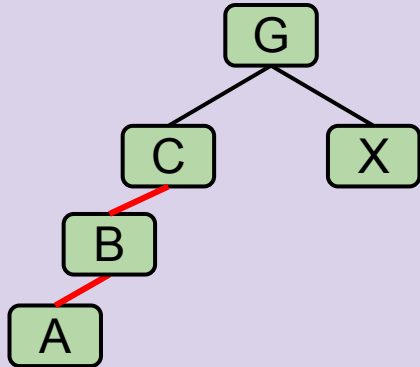
Searching an LLRB tree for a key is easy.

- Treat it exactly like any BST.



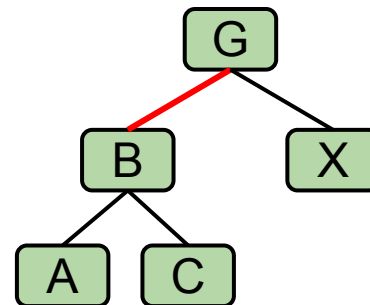
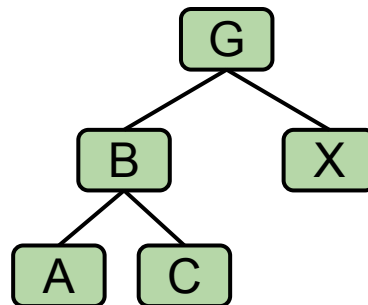
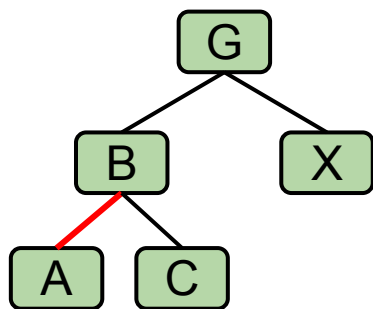
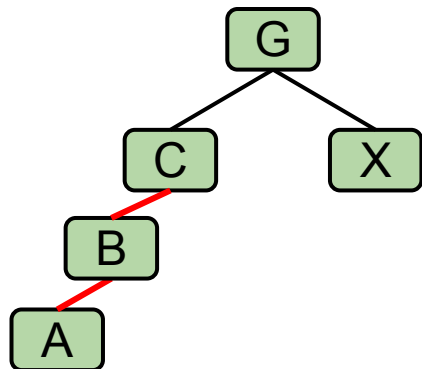
LLRB Problem #1: yellkey.com/person

How many of these are valid LLRBs, i.e. have a 1-1 correspondence with a valid 2-3 tree?

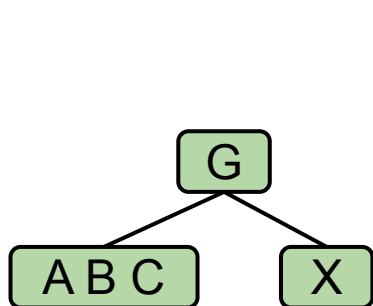


LLRB Problem #1: yellkey.com/person

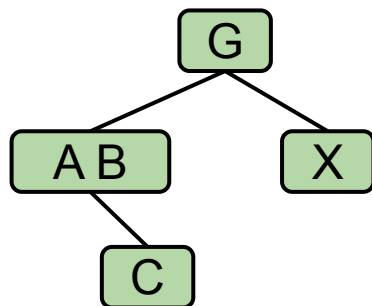
How many of these are valid LLRBs, i.e. have a 1-1 correspondence with a valid 2-3 tree?



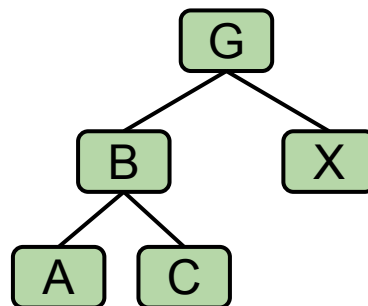
Equivalent 2-3



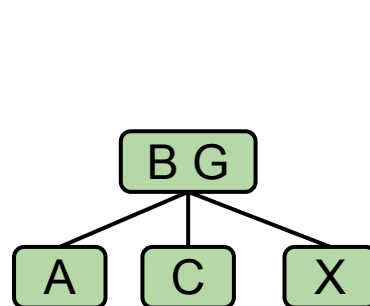
Invalid, has 4 node.



Invalid, not balanced.

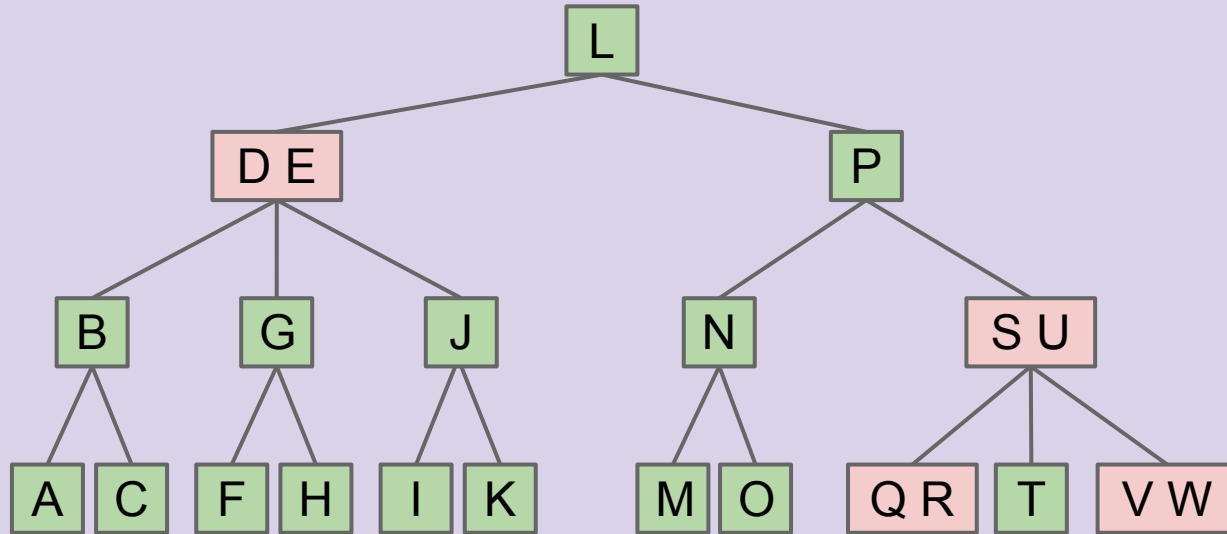


Invalid, not balanced.



LLRB Problem #2: yellkey.com/leave

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

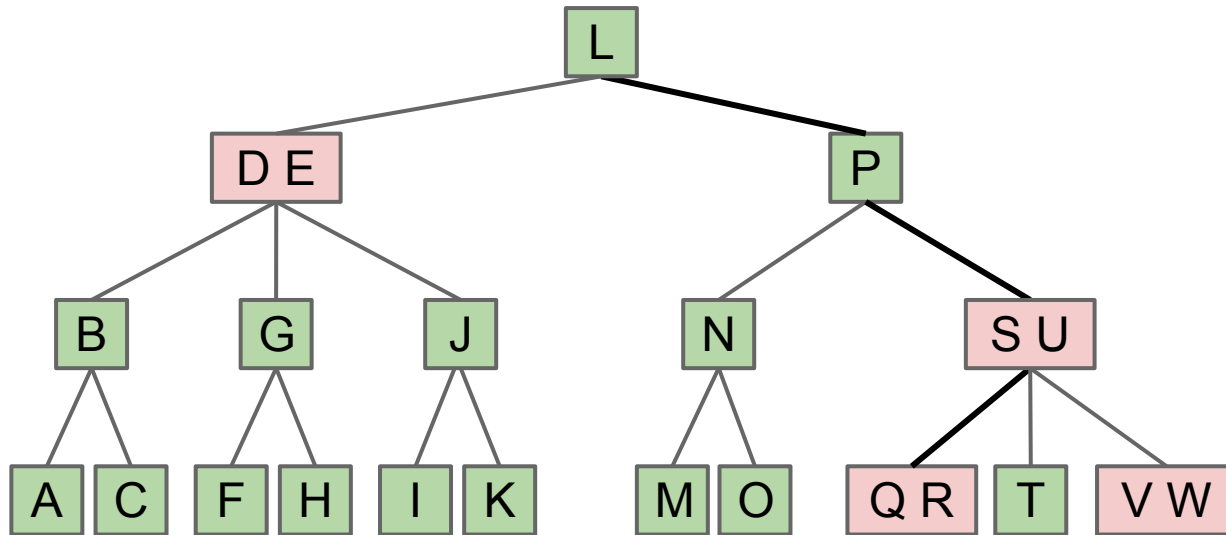


Note: The version of this slide used in live lecture was slightly different.

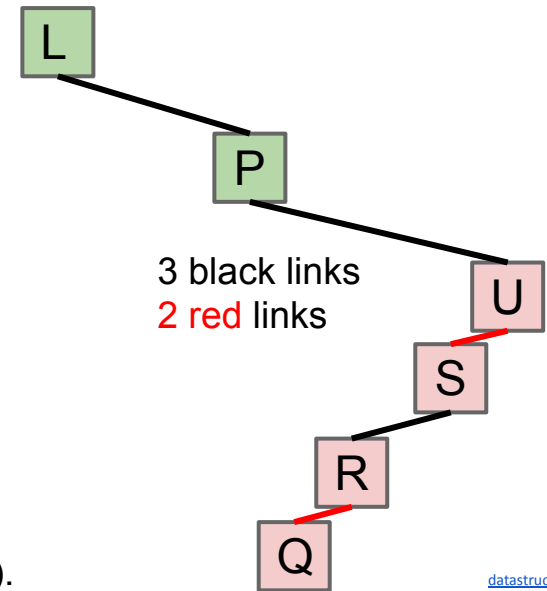
LLRB Problem #2: yellkey.com/leave

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

- 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 $\sim 2x$ the height of its 2-3 tree.



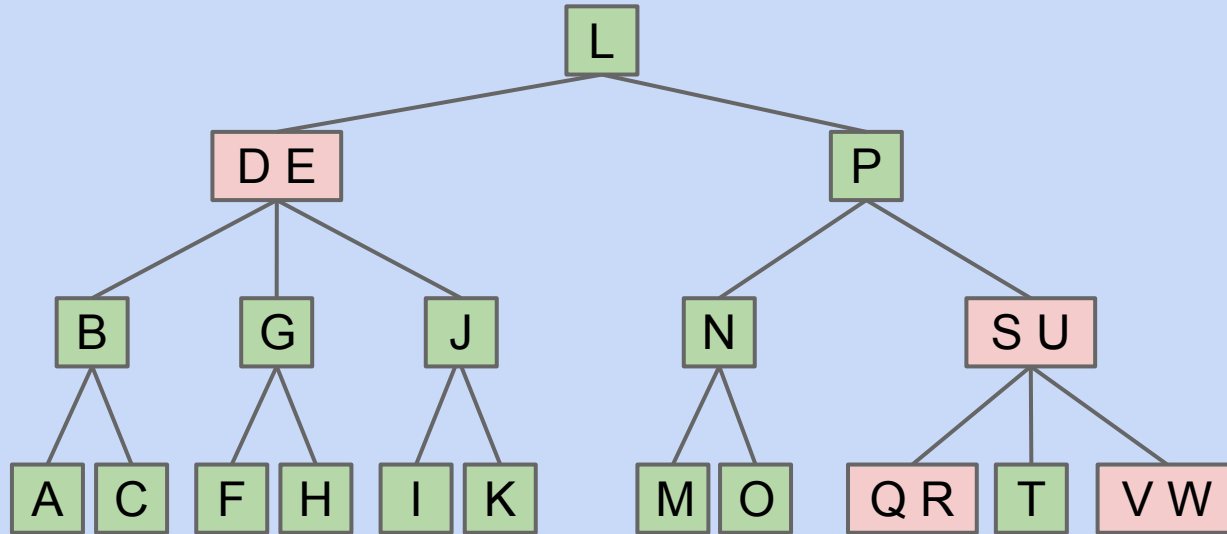
Dark line shows longest path (3 links).



LLRB Height

Suppose we have a 2-3 tree of height H.

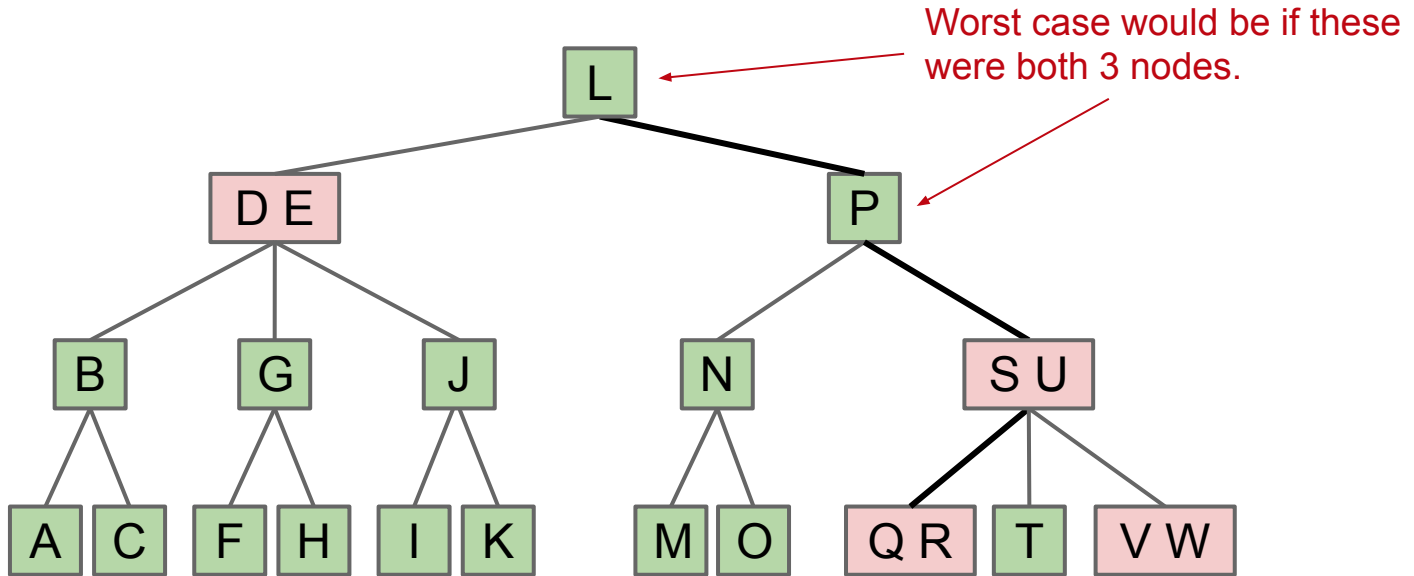
- What is the maximum height of the corresponding LLRB?



LLRB Height

Suppose we have a 2-3 tree of height H.

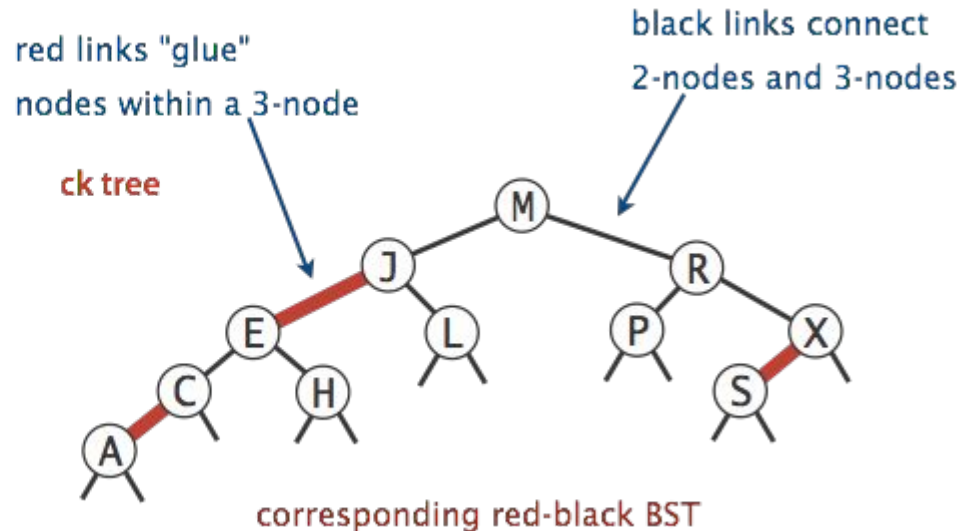
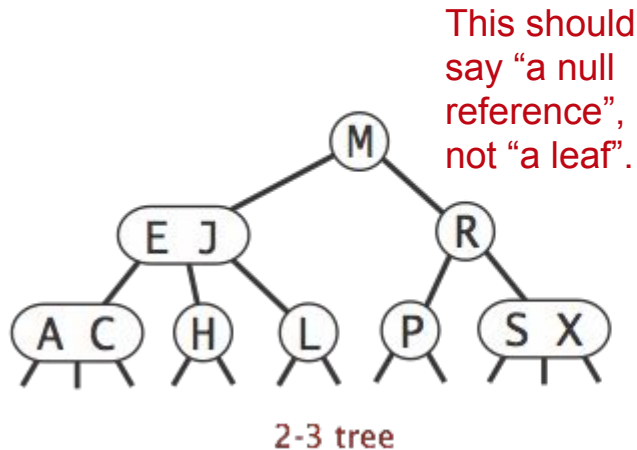
- What is the maximum height of the corresponding LLRB?
 - Total height is H (black) + **H + 1 (red)** = 2H + 1.



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

Some handy LLRB properties:

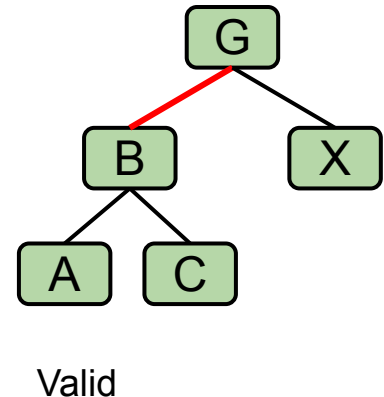
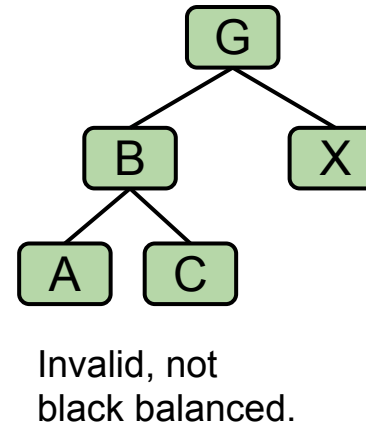
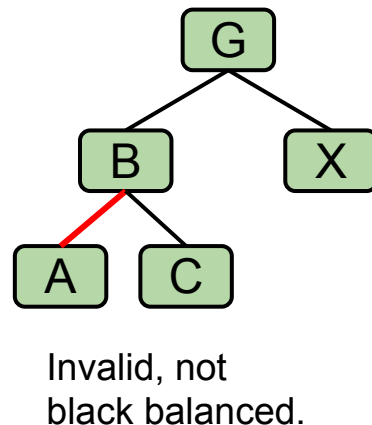
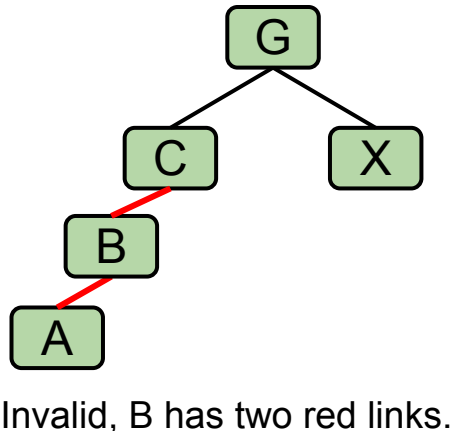
- No node has two red links [otherwise it'd be analogous to a 4 node, which are disallowed in 2-3 trees].
- Every path from root to a leaf has same number of **black links** [because 2-3 trees have the same number of links to every leaf]. LLRBs are therefore balanced.



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

Some handy LLRB properties:

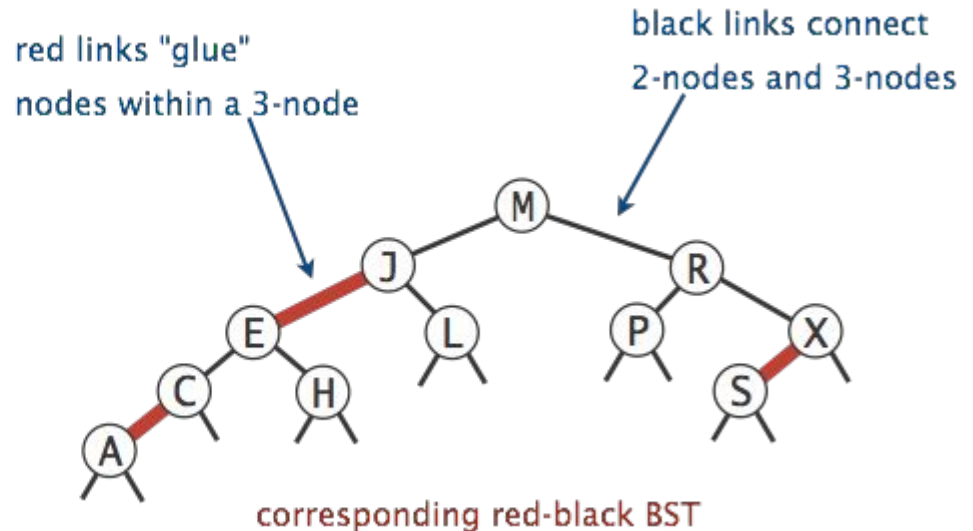
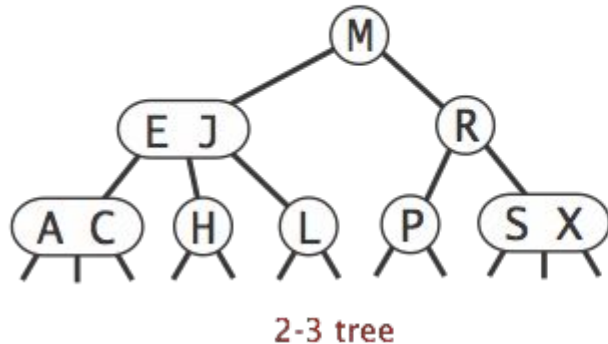
- No node has two red links [otherwise it'd be analogous to a 4 node, which are disallowed in 2-3 trees].
- Every path from root to a leaf has same number of **black links** [because 2-3 trees have the same number of links to every leaf]. LLRBs are therefore balanced.



LLRB Construction

One last important question: Where do LLRBs come from?

- Would not make sense to build a 2-3 tree, then convert. Even more complex.
- Instead, it turns out we implement an LLRB insert as follows:
 - Insert as usual into a BST.
 - Use zero or more rotations to maintain the 1-1 mapping.

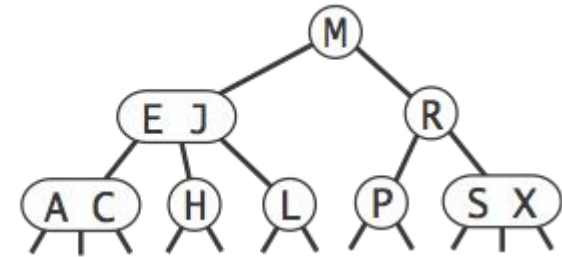


Maintaining 1-1 Correspondence Through Rotations

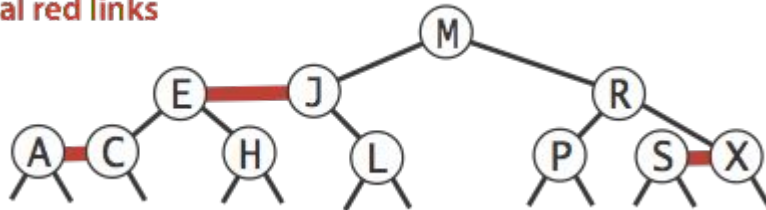
The 1-1 Mapping

There exists a 1-1 mapping between: **2-3 tree**

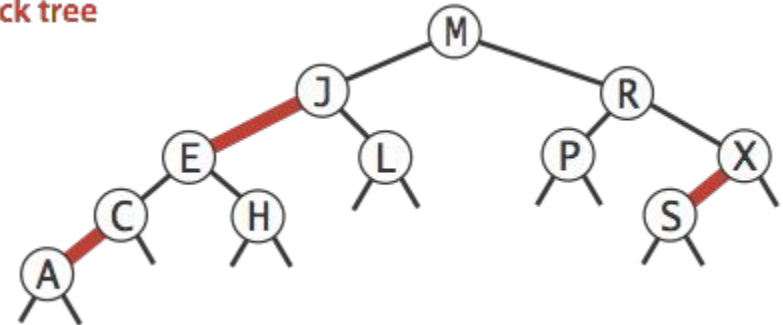
- 2-3 Tree
- LLRB



horizontal red links



red-black tree

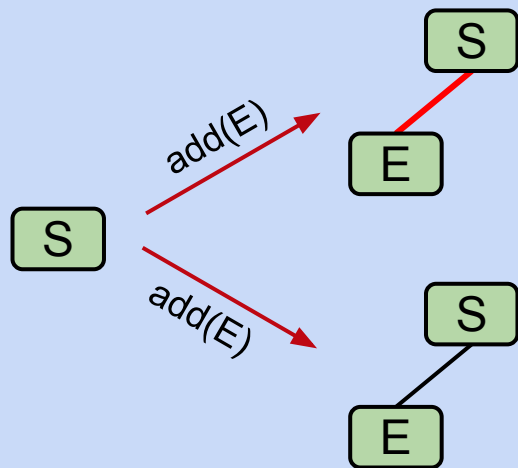


Implementation of an LLRB is based on maintaining this 1-1 correspondence.

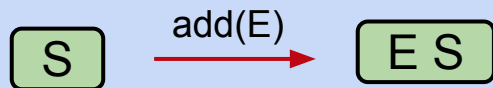
- When performing LLRB operations, pretend like you're a 2-3 tree.
- Preservation of the correspondence will involve tree rotations.

Design Task #1: Insertion Color

Should we use a red or black link when inserting?



LLRB World

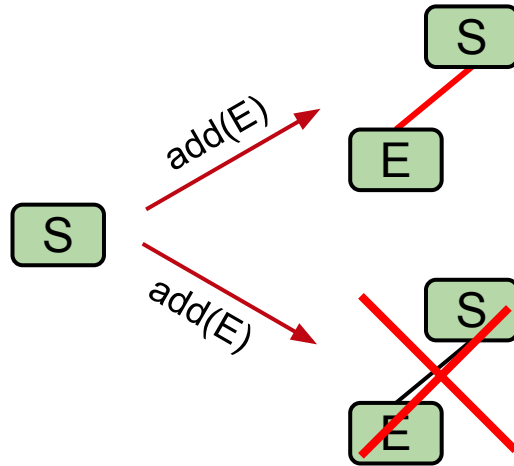


World 2-3

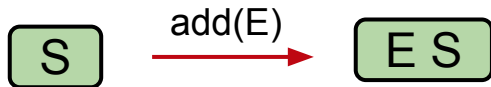
Design Task #1: Insertion Color

Should we use a red or black link when inserting?

- Use red! In 2-3 trees new values are ALWAYS added to a leaf node (at first).



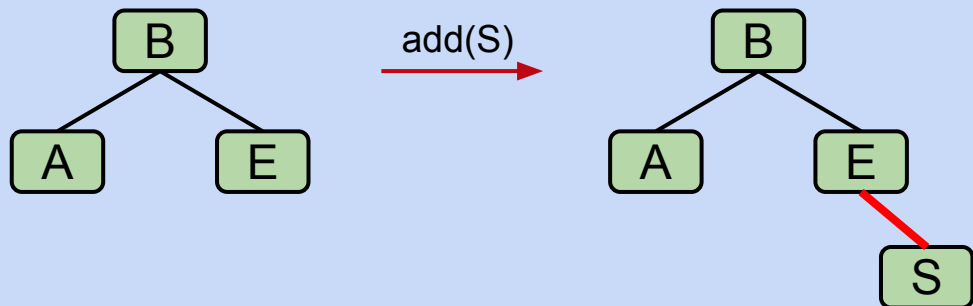
LLRB World



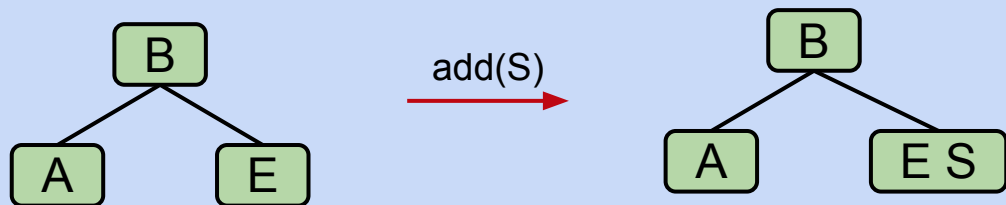
World 2-3

Design Task #2: Insertion on the Right

Suppose we have leaf E, and insert S with a red link. What is the problem below, and what do we do about it?



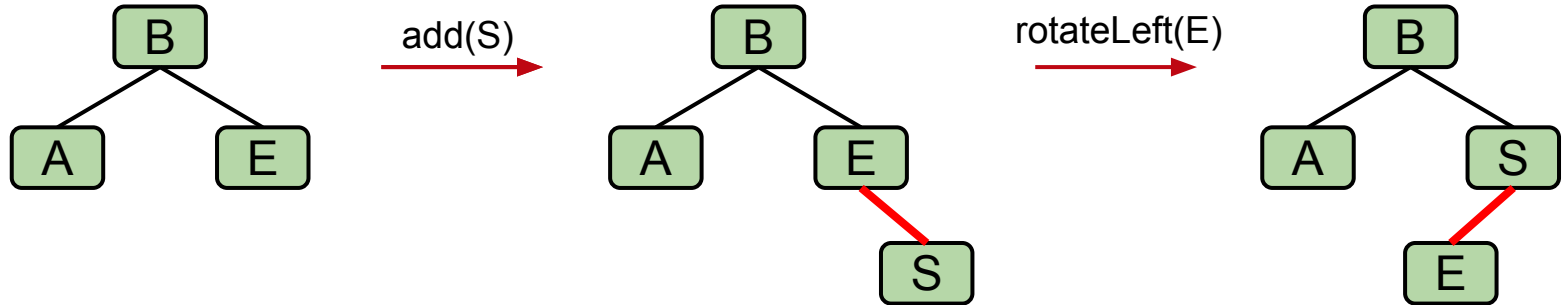
LLRB World



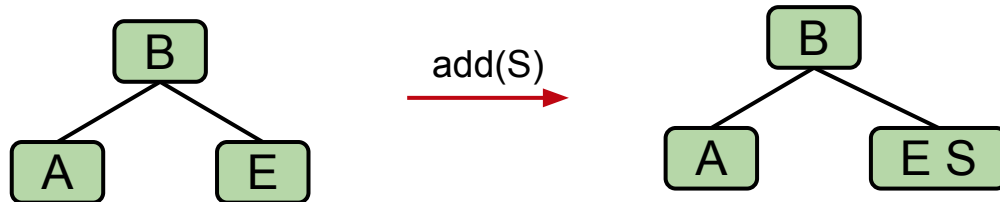
World 2-3

Design Task #2: Insertion on the Right

Suppose we have leaf E, and insert S with a red link. What is the problem below, and what do we do about it: Right links aren't allowed, so rotateLeft(E).



LLRB World

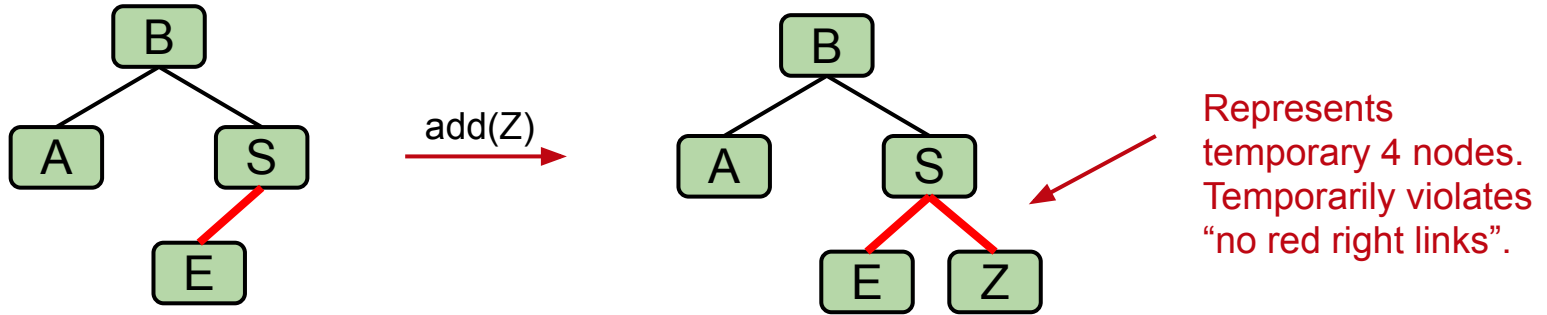


World 2-3

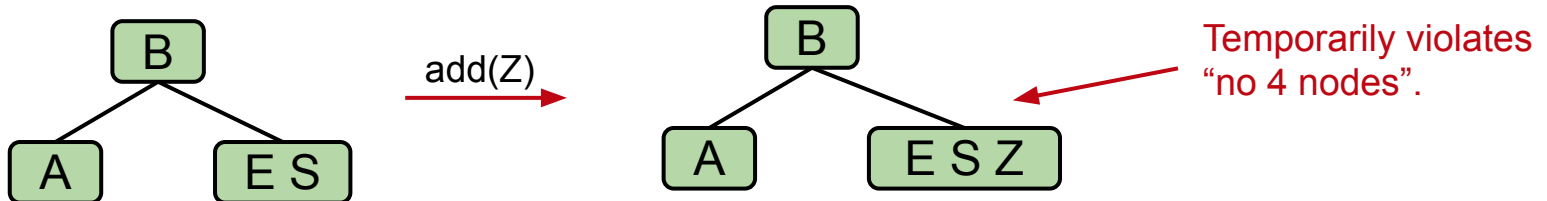
New Rule: Representation of Temporary 4-Nodes

We will represent temporary 4-nodes as BST nodes with two red links.

- This state is only temporary (more soon), so temporary violation of “left leaning” is OK.



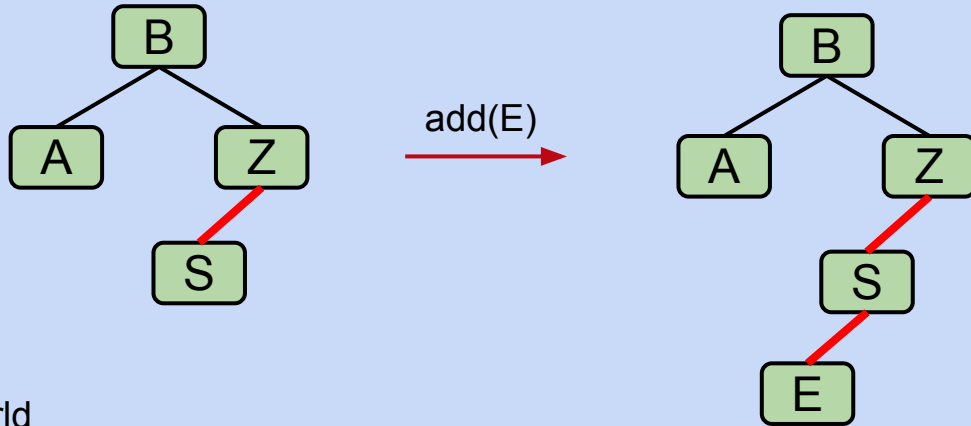
LLRB World



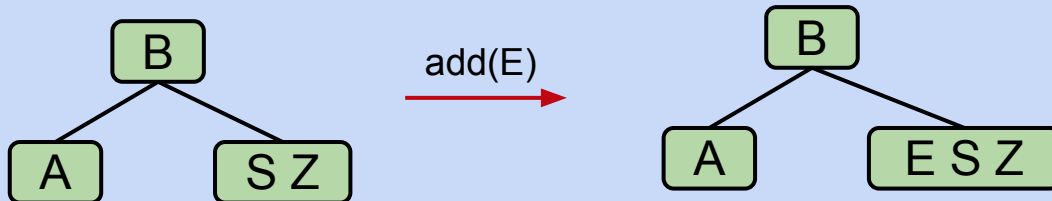
World 2-3

Design Task #3: Double Insertion on the Left

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?



LLRB World

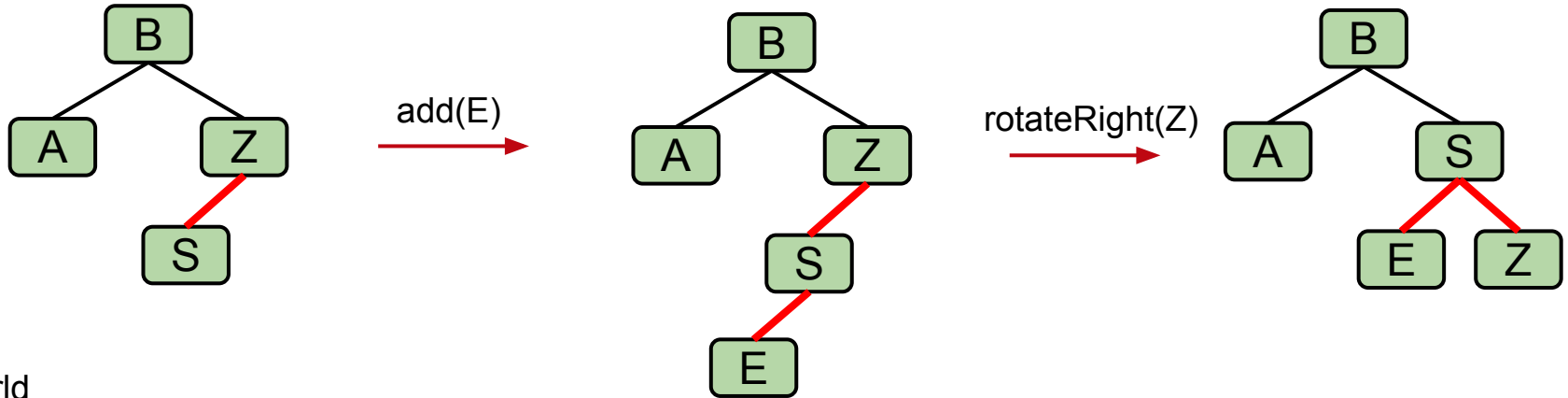


World 2-3

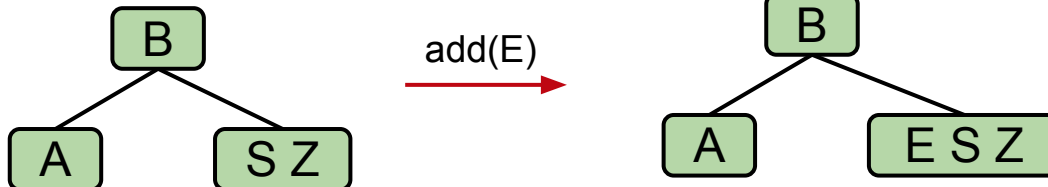
Design Task #3: Double Insertion on the Left

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?

- Rotate Z right.



LLRB World

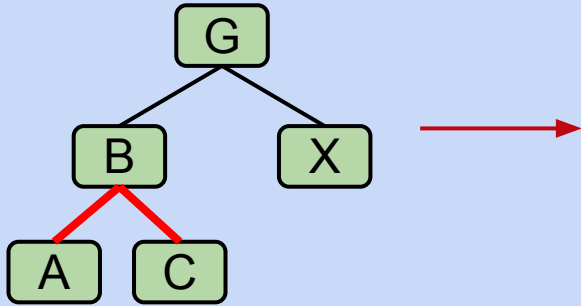


World 2-3

Design Task #4: Splitting Temporary 4-Nodes

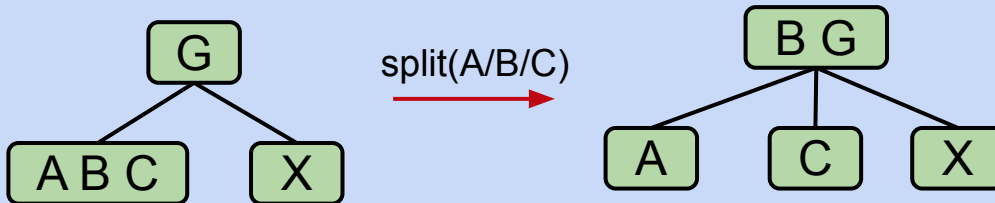
Suppose we have the LLRB below which includes a temporary 4 node.
What should we do next?

- Try to figure this one out! It's a particularly interesting puzzle.



Hint: Ask yourself “What Would 2-3 Tree Do?” WW23TD?

LLRB World

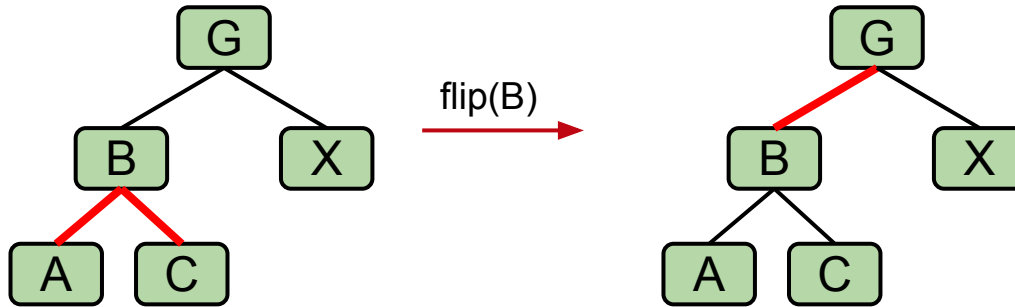


World 2-3

Design Task #4: Splitting Temporary 4-Nodes

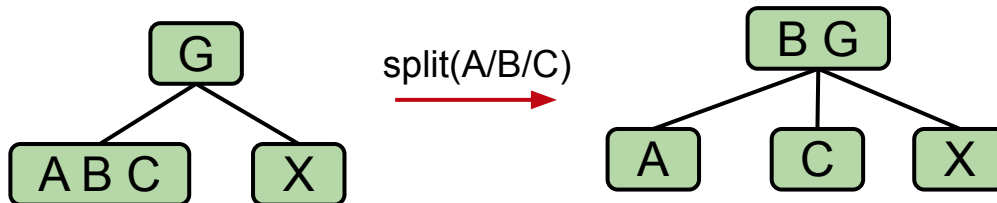
Suppose we have the LLRB below which includes a temporary 4 node.
What should we do next?

- Flip the colors of all edges touching B.
 - Note: This doesn't change the BST structure/shape.



BST, the magic was inside of you all along.

LLRB World



World 2-3

... and That's It!

Congratulations, you just invented the red-black BST.

- 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.

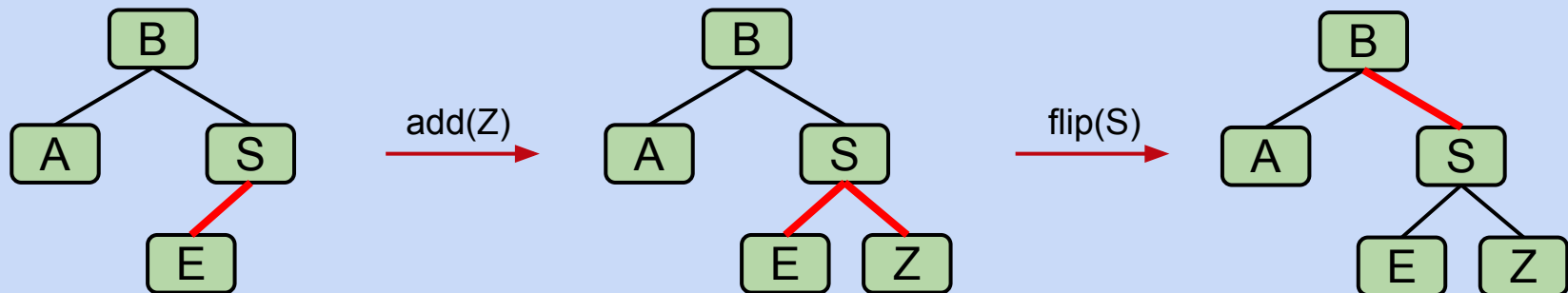
One last detail: Cascading operations.

- It is possible that a rotation or flip operation will cause an additional violation that needs fixing.

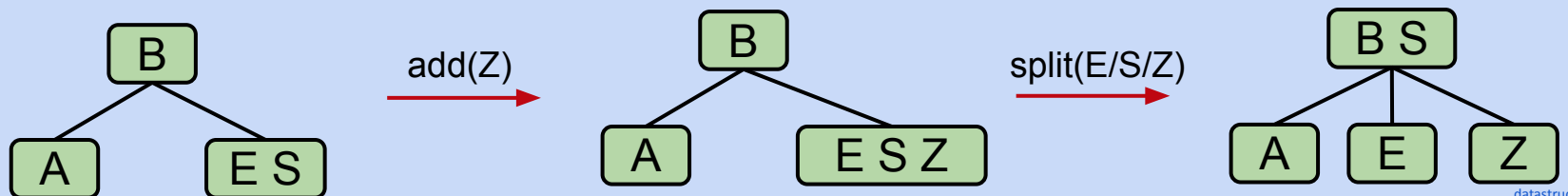
Cascading Balance Example

Inserting Z gives us a temporary 4 node.

- Color flip yields an invalid tree. Why? What's next?



LLRB World

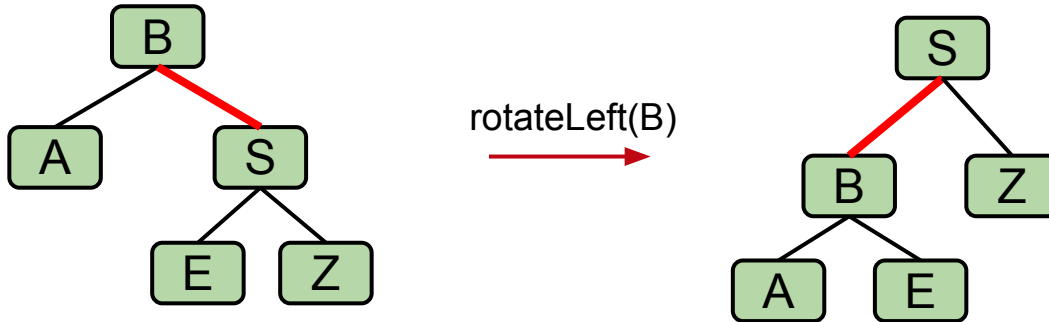


World 2-3

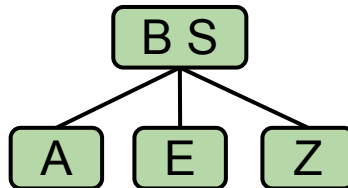
Cascading Balance Example

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

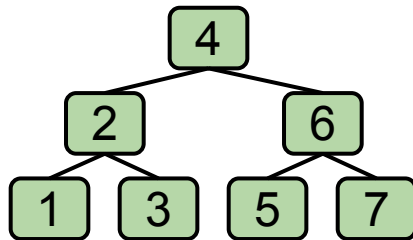
Optional Exercise

Insertion of 7 through 1

To get an intuitive understanding of why all this works, try inserting the 7, 6, 5, 4, 3, 2, 1, into an initially empty LLRB.

- You should end up with a perfectly balanced BST!

To check your work, see this [demo](#).



LLRB Runtime and Implementation

LLRB Runtime

The runtime analysis for LLRBs is simple if you trust the 2-3 tree runtime.

- LLRB tree has height $O(\log N)$.
- Contains is trivially $O(\log N)$.
- Insert is $O(\log N)$.
 - $O(\log N)$ to add the new node.
 - $O(\log N)$ rotation and color flip operations per insert.

We will not discuss LLRB delete.

- Not too terrible really, but it's just not interesting enough to cover. See optional textbook if you're curious (though they gloss over it, too).

LLRB Implementation

Amazingly, turning a BST into an LLRB requires only 3 clever lines of code.

- Does not include helper methods (which do not require cleverness).

```
private Node put(Node h, Key key, Value val) {
    if (h == null) { return new Node(key, val, RED); }

    int cmp = key.compareTo(h.key);
    if (cmp < 0) { h.left = put(h.left, key, val); }
    else if (cmp > 0) { h.right = put(h.right, key, val); }
    else { h.val = val; }

    if (isRed(h.right) && !isRed(h.left)) { h = rotateLeft(h); }
    if (isRed(h.left) && isRed(h.left.left)) { h = rotateRight(h); }
    if (isRed(h.left) && isRed(h.right)) { flipColors(h); }

    return h;
}
```

} just 3 lines
needed
to balance

Search Tree Summary

Search Trees

In the last 3 lectures, we talked about using search trees to implement sets/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.

... and Beyond

There are many other types of search trees out there.

- Other self balancing trees: 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.