Lecture 18 (Data Structures 4)

Tree Rotation and Red-Black Trees

CS61B, Spring 2024 @ UC Berkeley

Slides credit: Josh Hug

B-Trees Are Ugly to Implement

Lecture 18, CS61B, Spring 2024

The Bad News

B-Trees for small L, e.g. 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.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

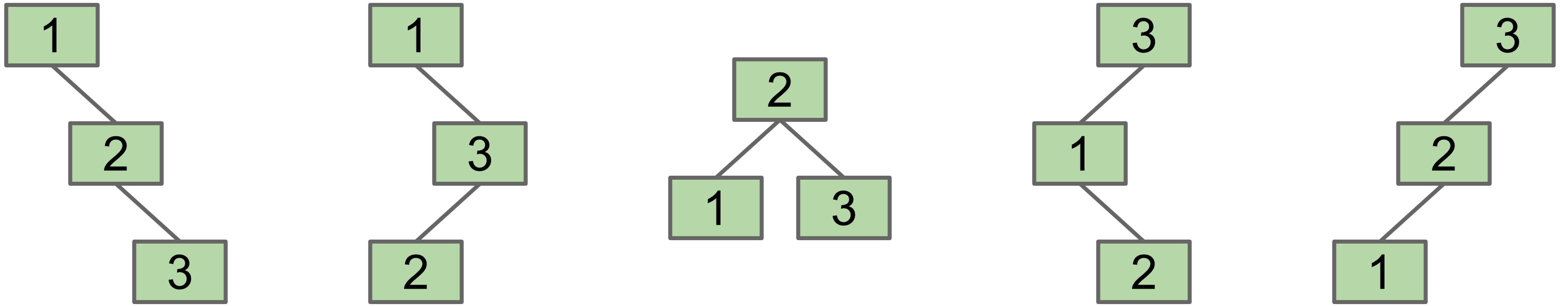
Definition of Tree Rotation

Lecture 18, CS61B, Spring 2024

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.



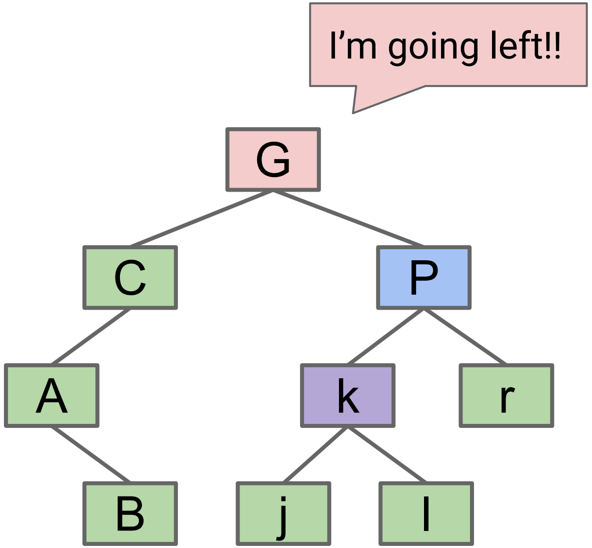
Given any BST, it is possible to move to a different configuration using “rotation”.

- 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 (Demo)

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

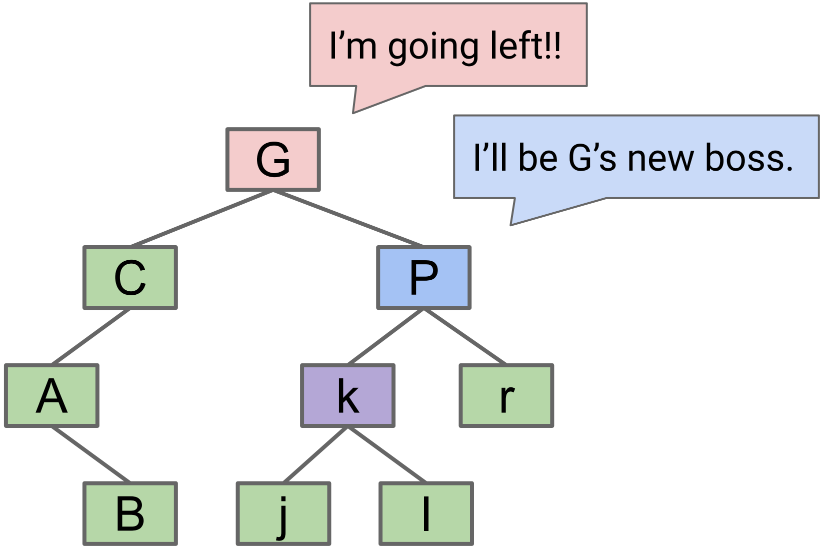
- Preserves search tree property. No change to semantics of tree.



Tree Rotation Definition (Demo)

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

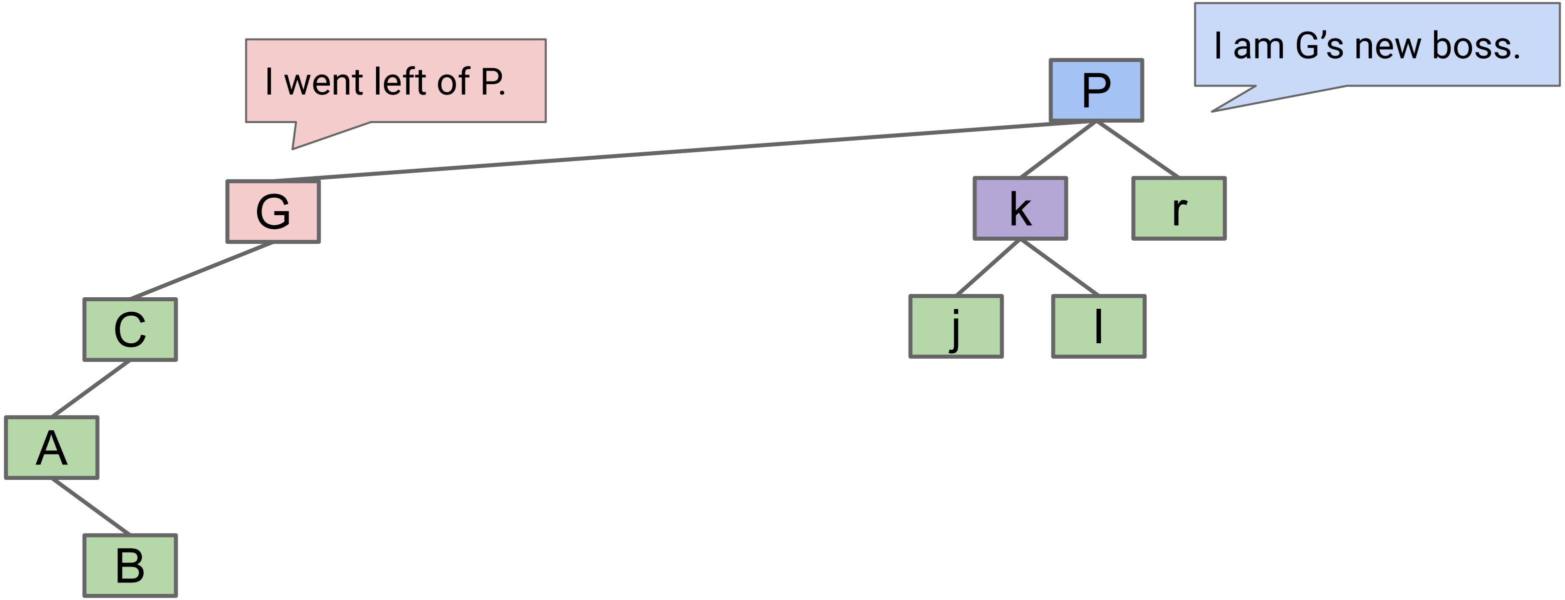
- Preserves search tree property. No change to semantics of tree.



Tree Rotation Definition (Demo)

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

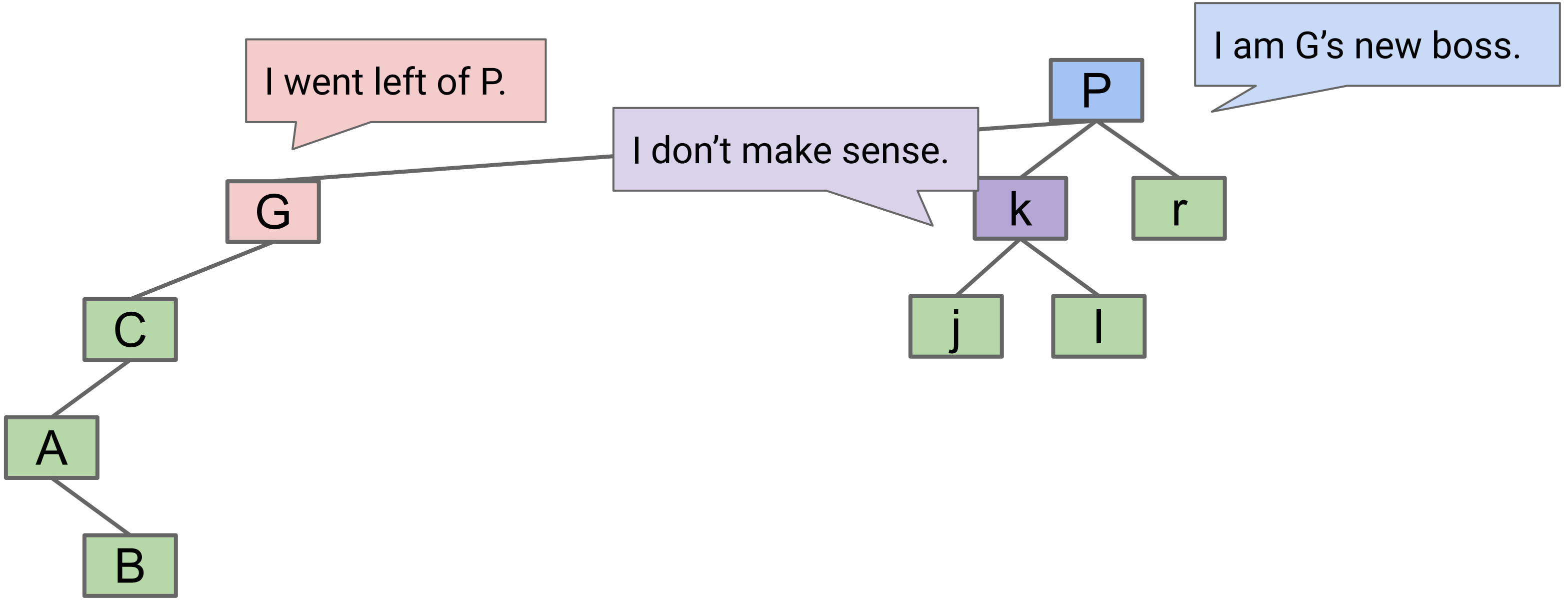
- Preserves search tree property. No change to semantics of tree.



Tree Rotation Definition (Demo)

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

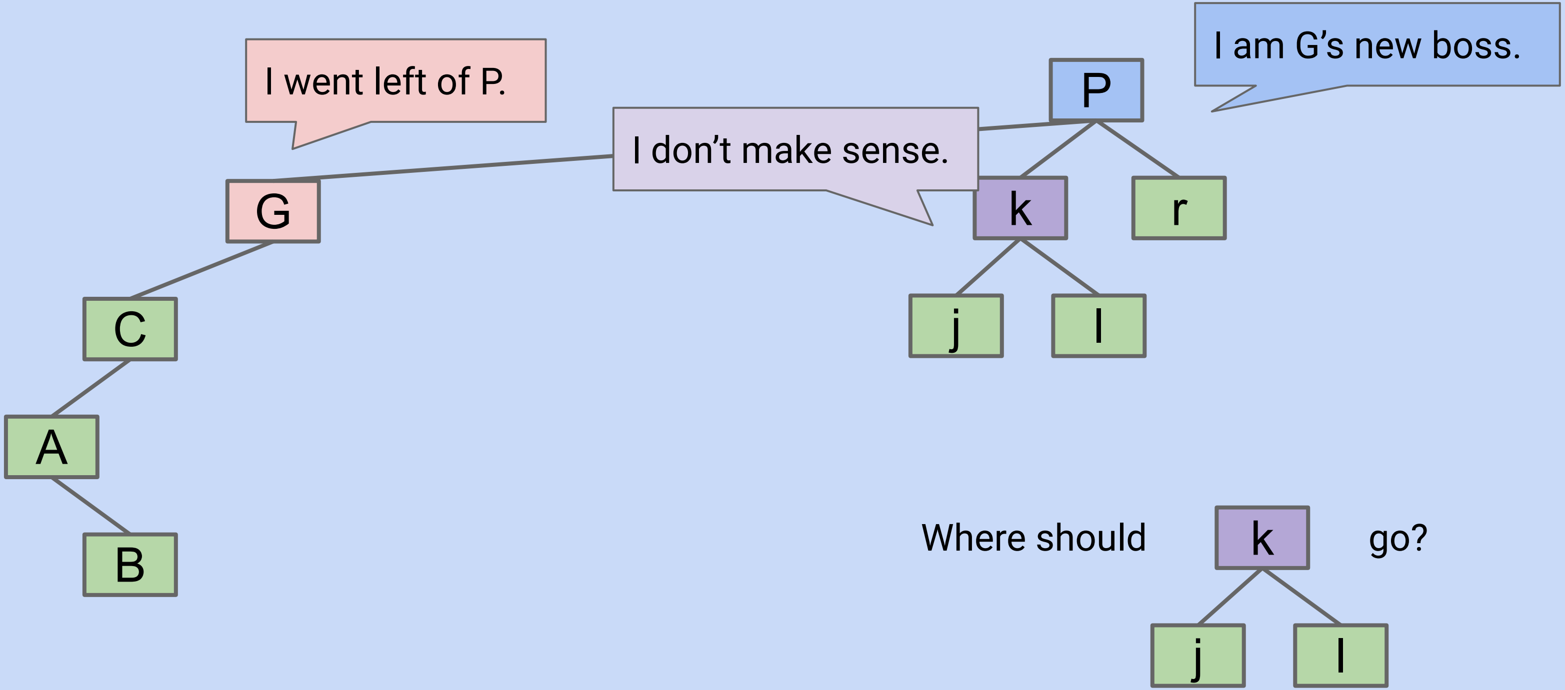
- Preserves search tree property. No change to semantics of tree.



Tree Rotation Definition (Demo)

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

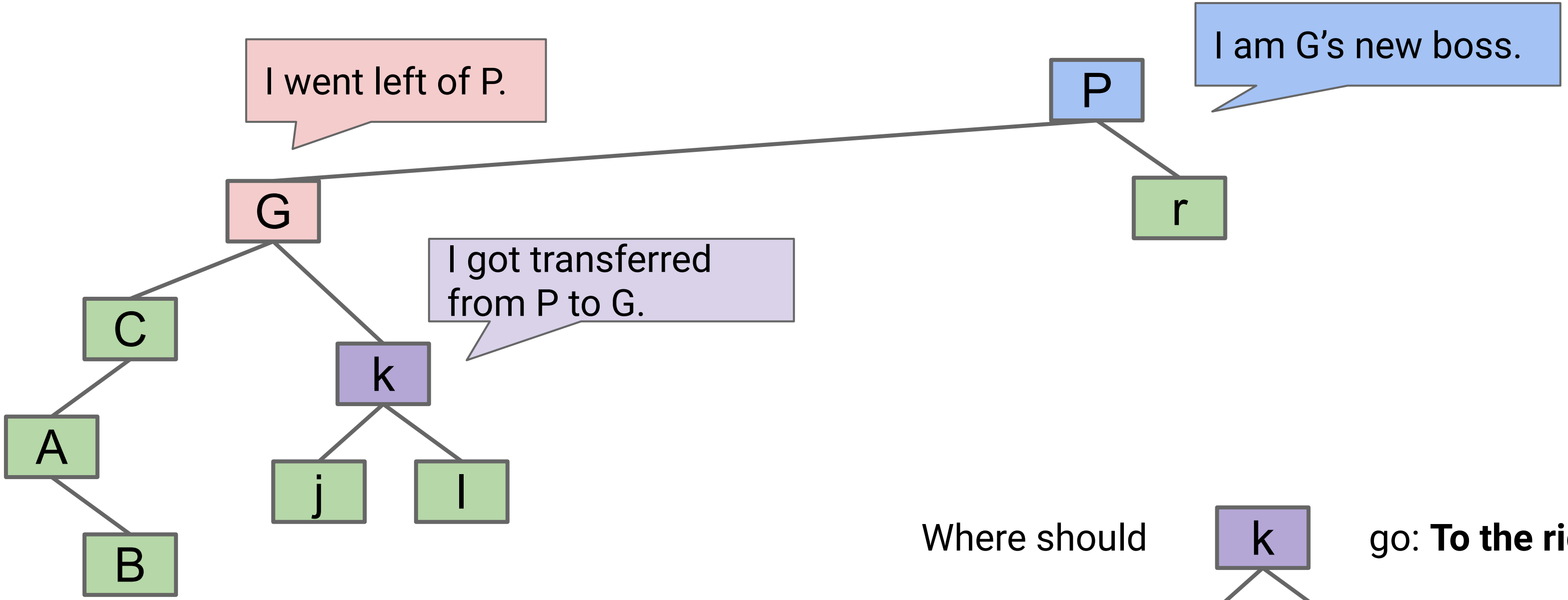
- Preserves search tree property. No change to semantics of tree.



Tree Rotation Definition (Demo)

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.

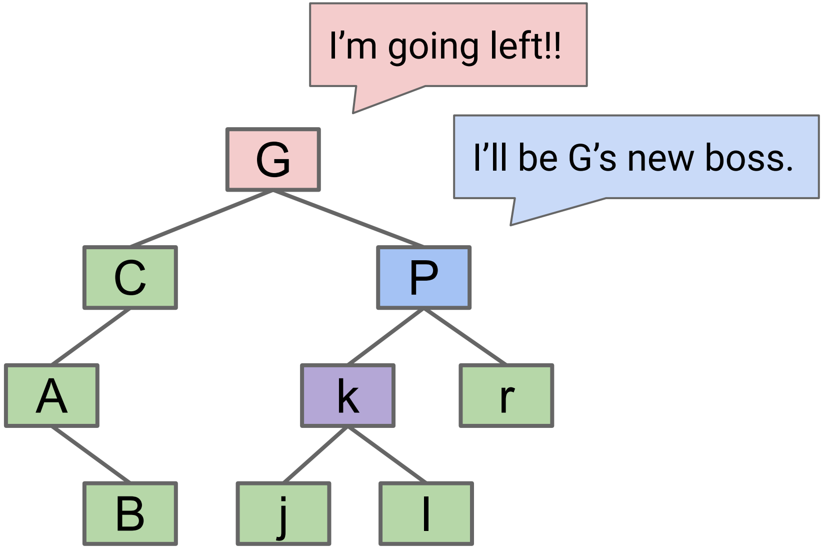


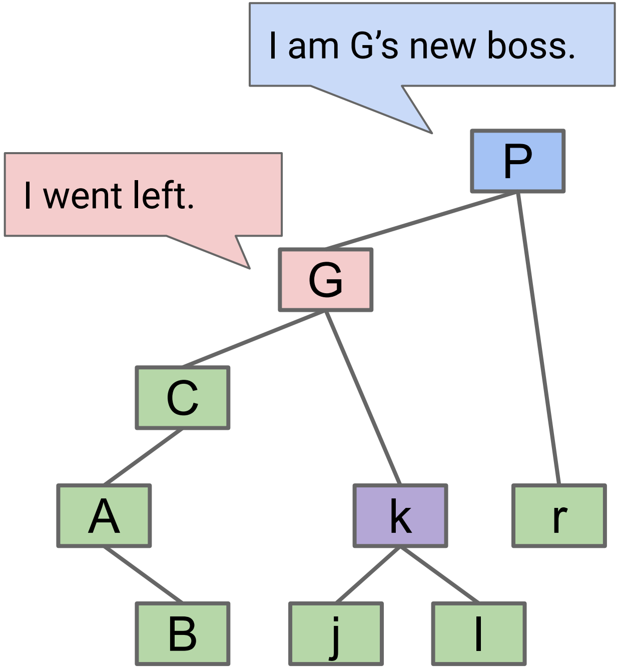
Where should k go: **To the right of G.**

Tree Rotation Definition (All in One Slide)

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.



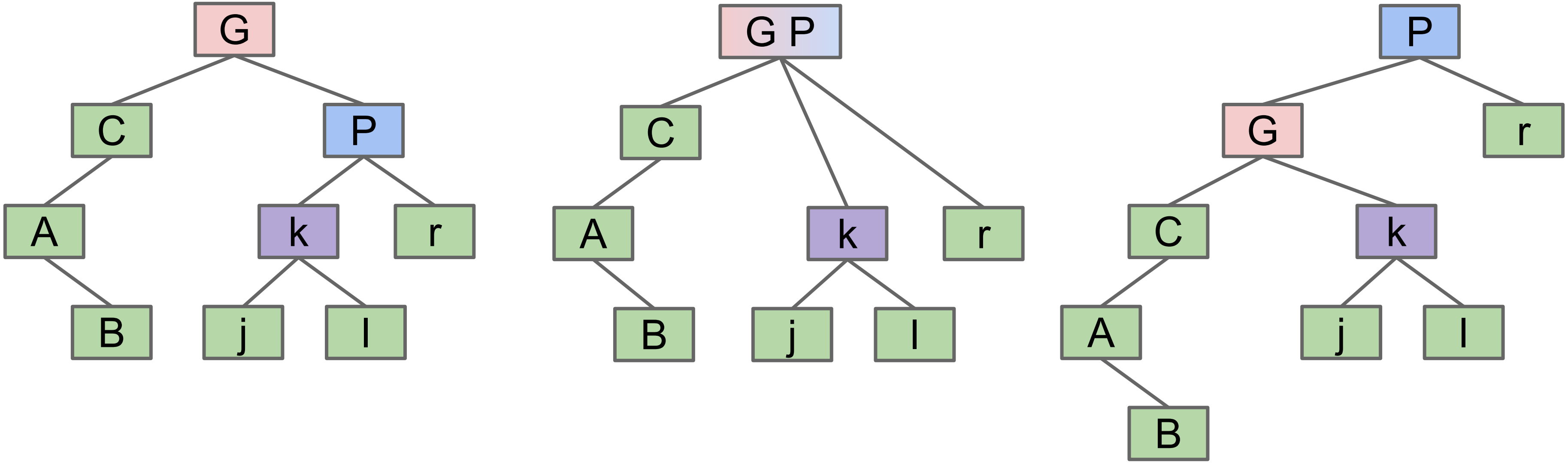


For this example rotateLeft(G) increased height of tree!

Tree Rotation Definition (Alternate Interpretation)

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.

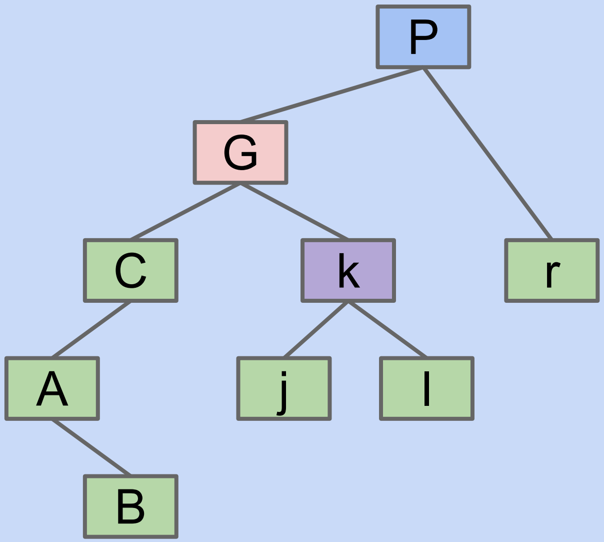


For this example rotateLeft(G) increased height of tree!

Your Turn

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

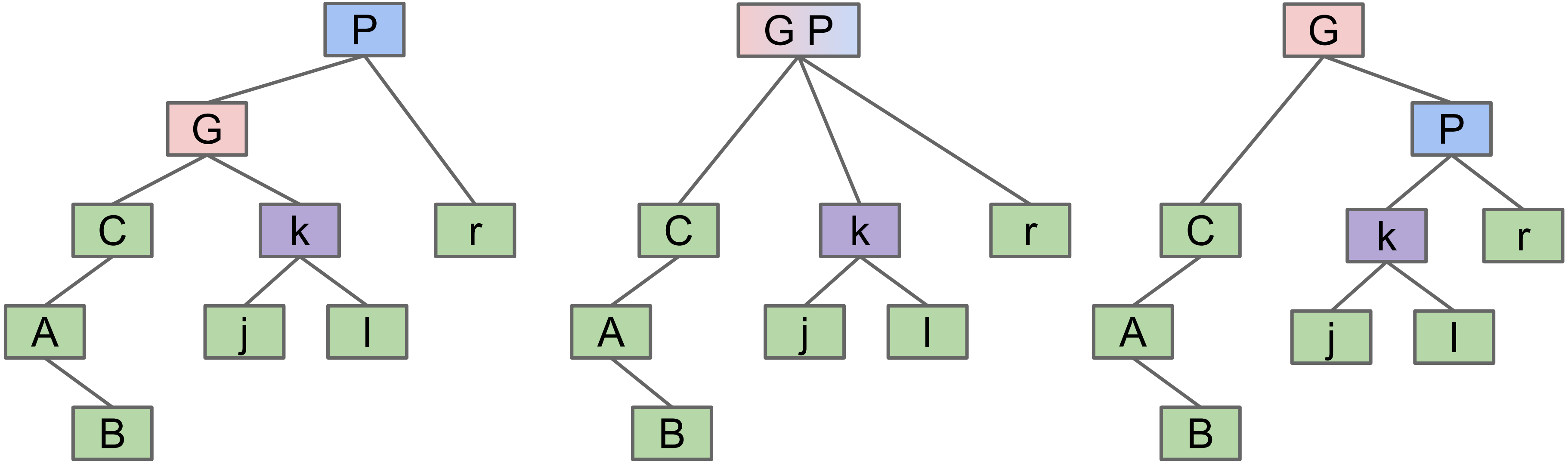
- Can think of as temporarily merging G and P, then sending P down and **right**.



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**.
- Note: k was G’s right child. Now it is P’s left child.

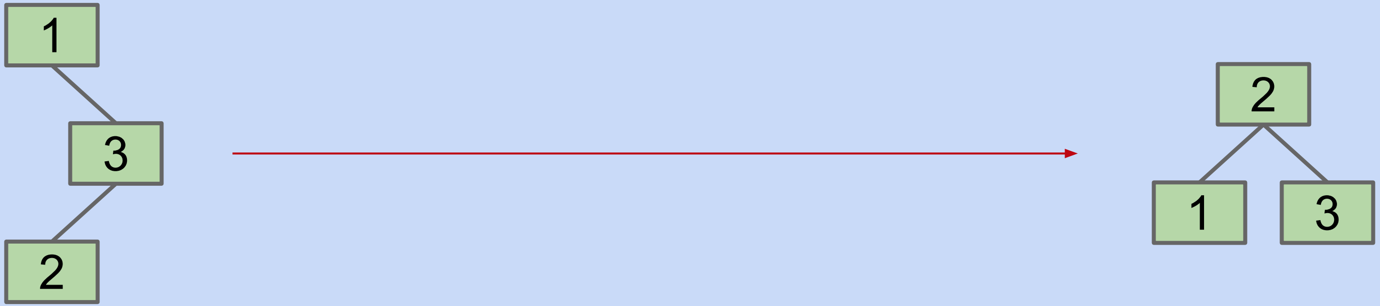


For this example rotateRight(P) decreased height of tree!

Tree Balancing

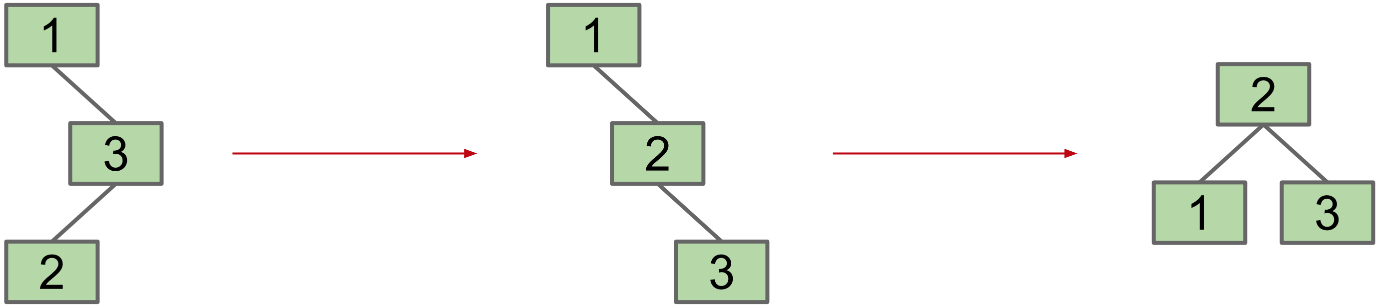
Lecture 18, CS61B, Spring 2024

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



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

- rotateRight(3)
- rotateLeft(1)

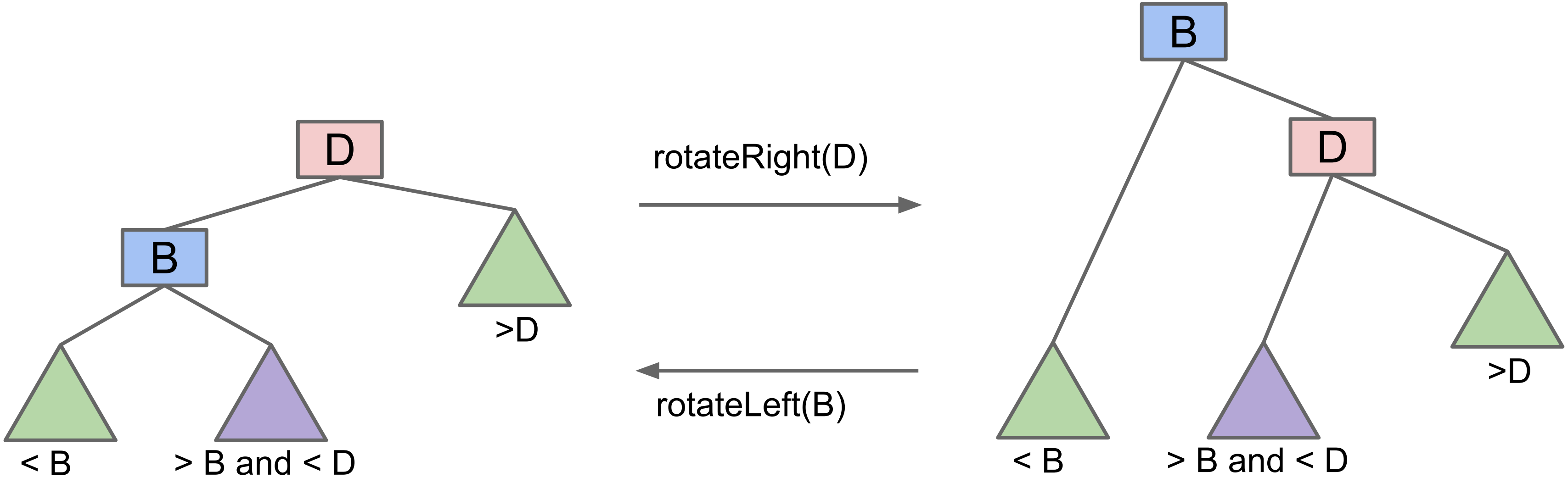


There are other correct answers as well!

Rotation for Balance

Rotation:

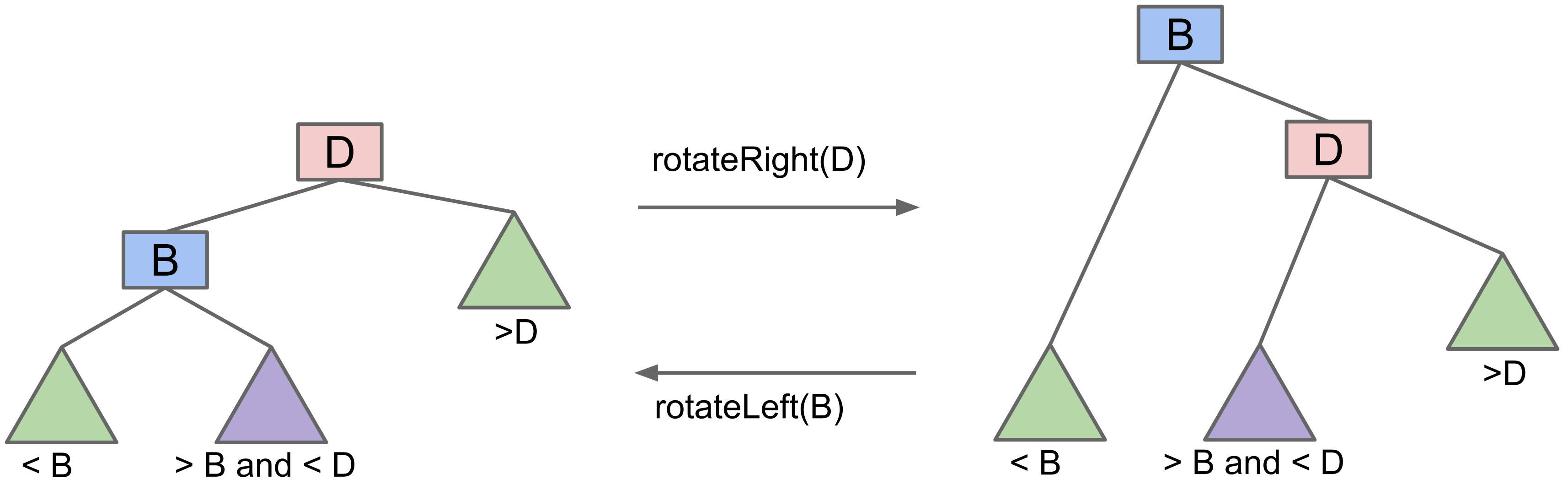
- Can shorten (or lengthen) a tree.
- Preserves search tree property.



Rotation for Balance

Rotation:

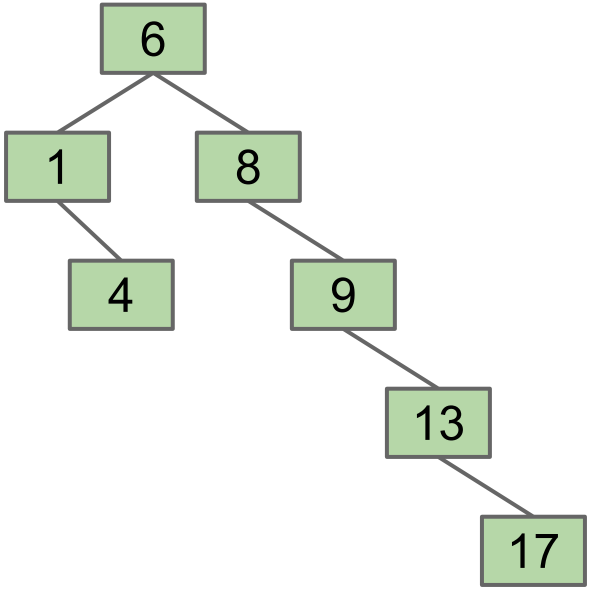
- Can shorten (or lengthen) a tree.
- Preserves search tree property.



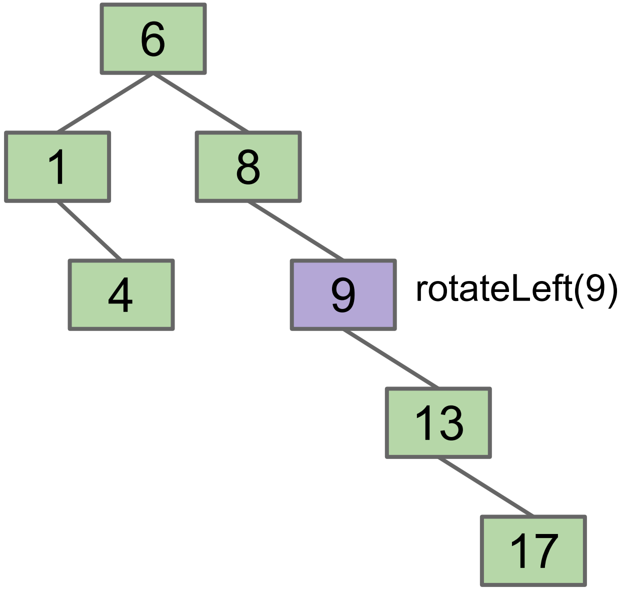
Can use rotation to balance a BST.

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

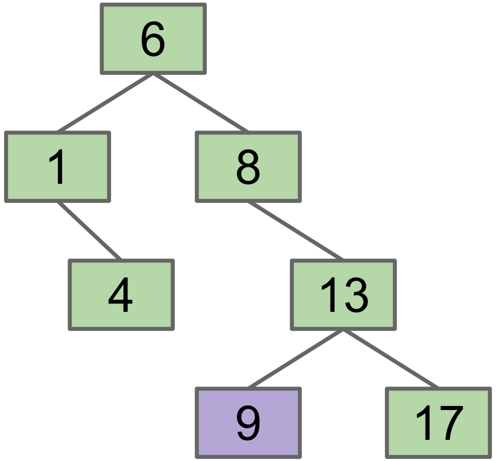
Demo: Balancing with Tree Rotation



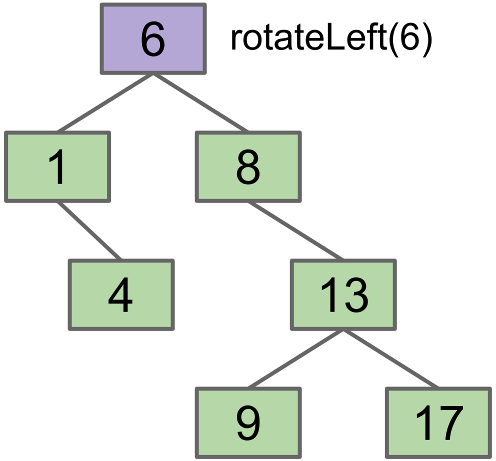
Demo: Balancing with Tree Rotation



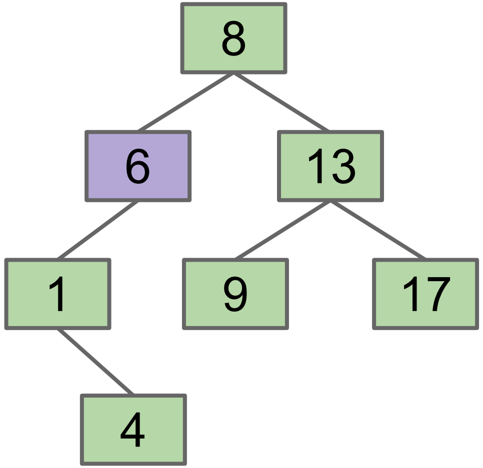
Demo: Balancing with Tree Rotation



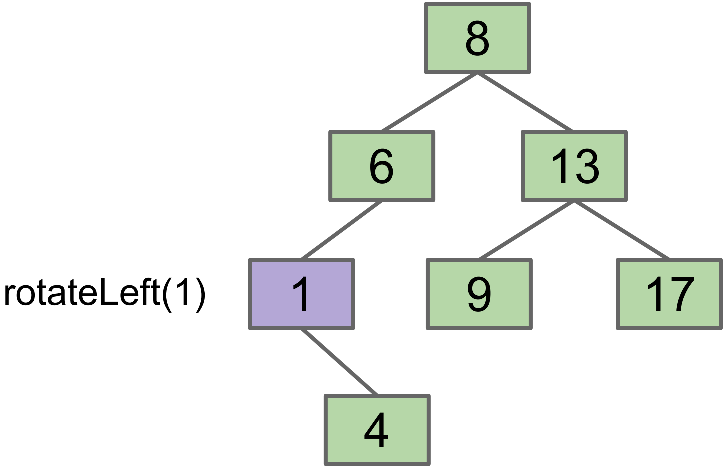
Demo: Balancing with Tree Rotation



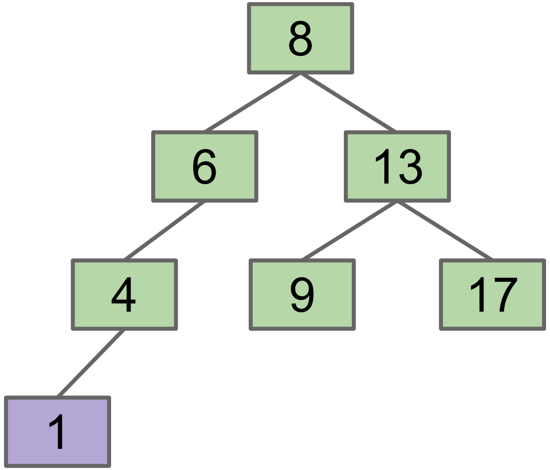
Demo: Balancing with Tree Rotation



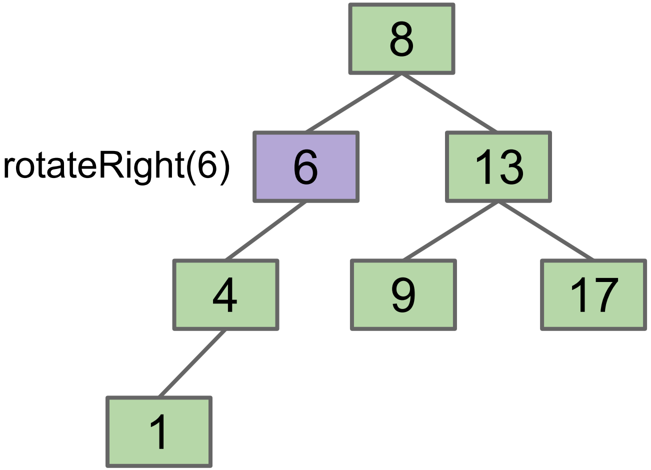
Demo: Balancing with Tree Rotation



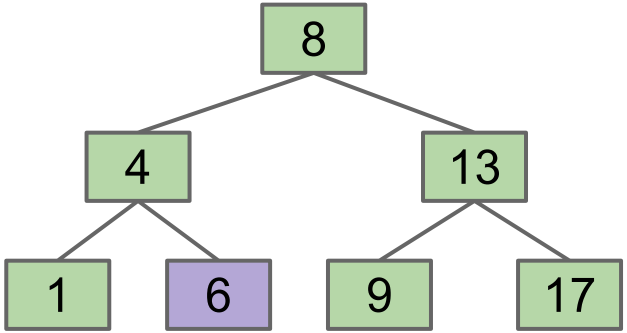
Demo: Balancing with Tree Rotation



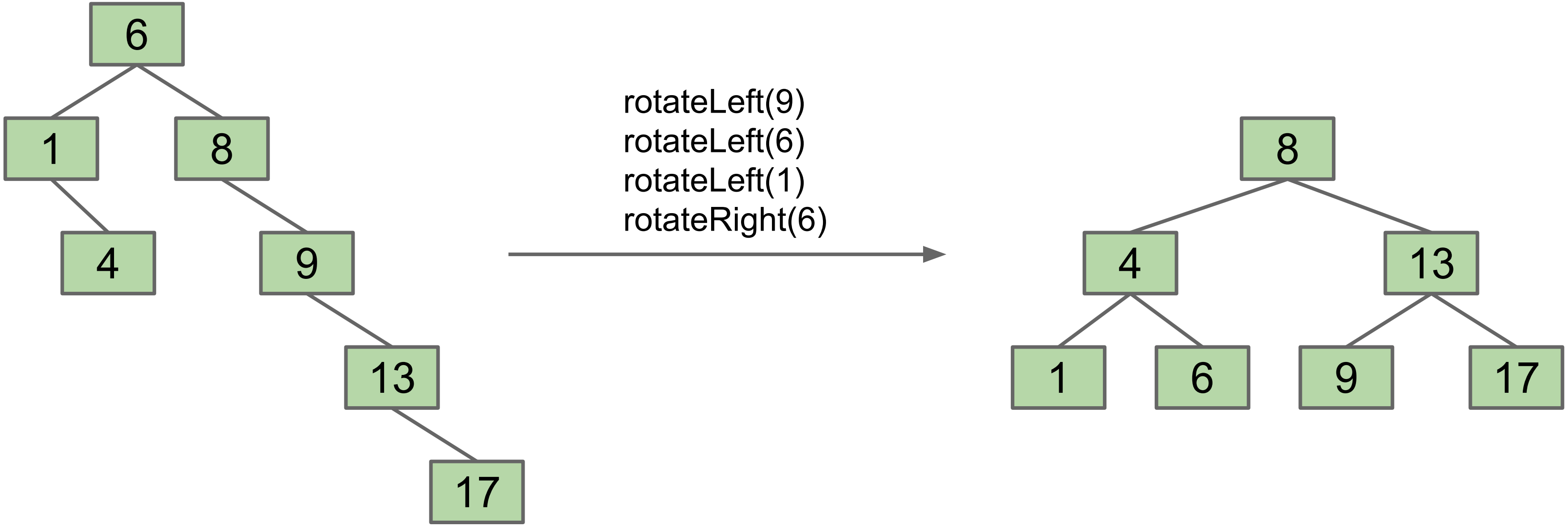
Demo: Balancing with Tree Rotation



Demo: Balancing with Tree Rotation

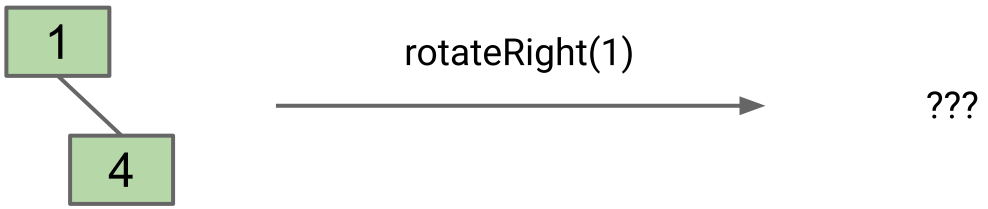


Demo: Balancing with Tree Rotation



Some Rotations are Undefined

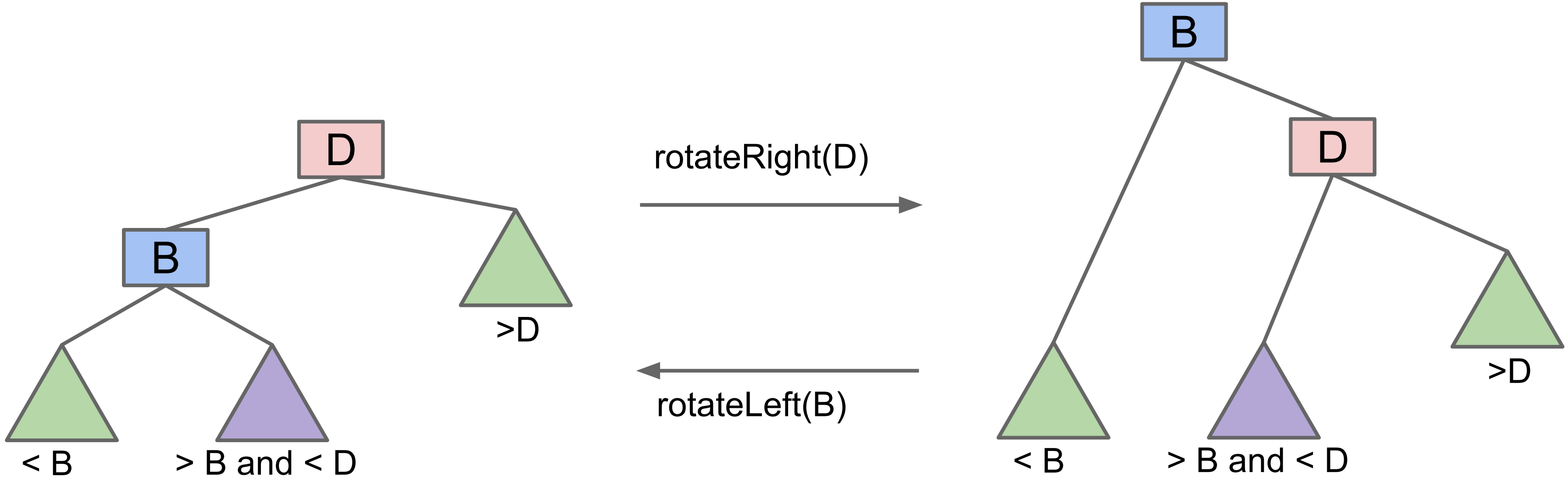
- Rotating a node right is undefined if that node has no left child.
  - We would need to promote that node's left child, but it doesn't exist.
- Rotating a node left is undefined if that node has no right child.
- We won't need to perform any undefined rotations in this lecture, so don't worry about them.



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...

The 2-3 Tree Isometry

Lecture 18, CS61B, Spring 2024

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.

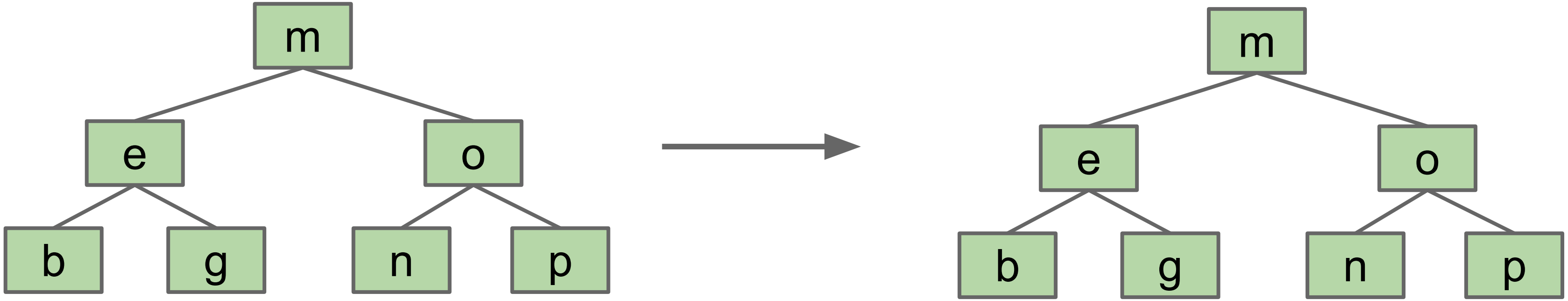
Our goal: Build a BST that is structurally identical to a 2-3 tree.

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

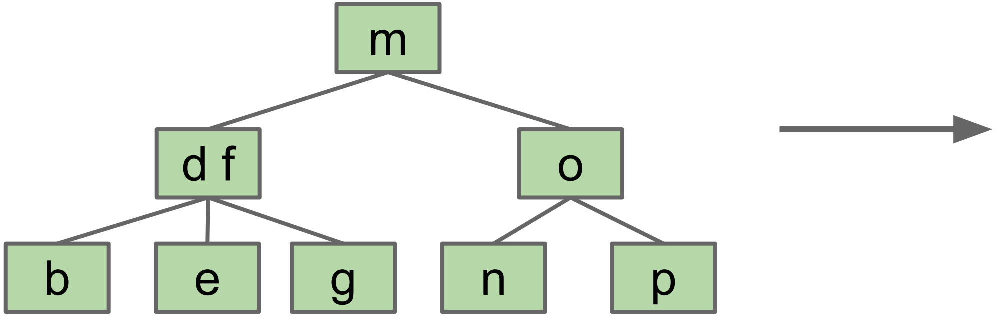
Representing a 2-3 Tree as a BST

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

- BST is exactly the same!



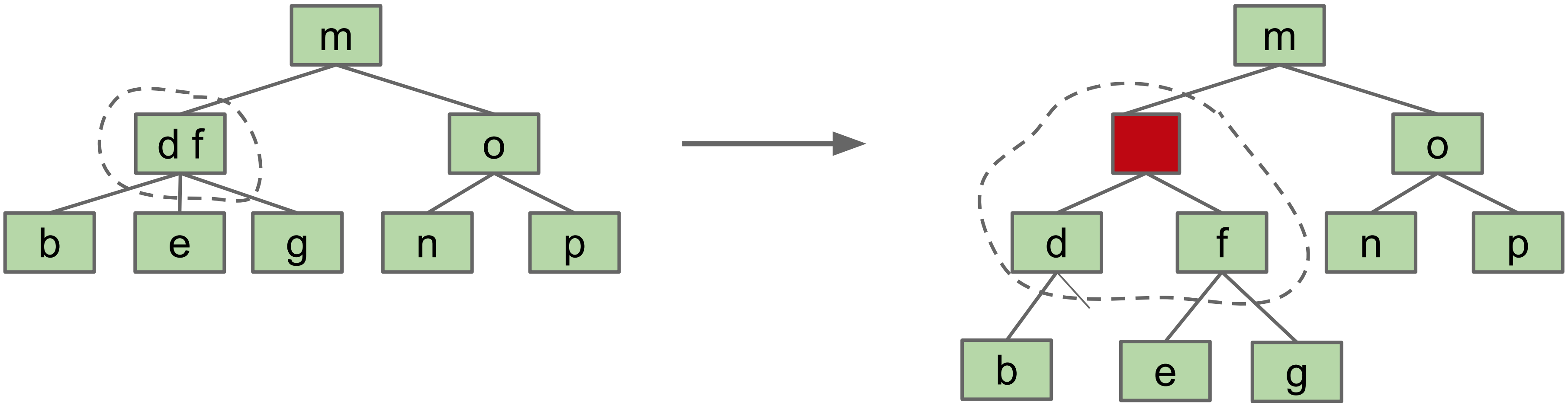
What do we do about 3-nodes?



????

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

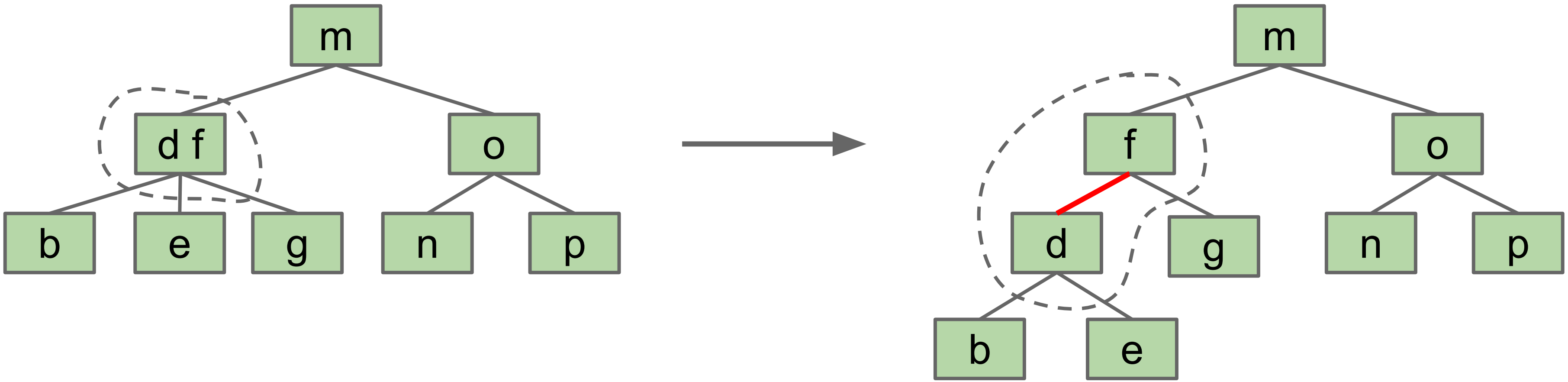
Possibility 1: Create dummy “glue” nodes.



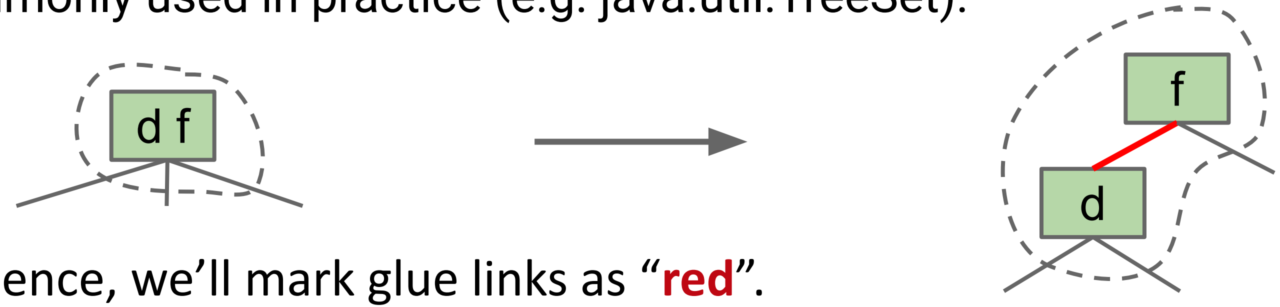
Result is inelegant. Wasted link. Code will be ugly.

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

Possibility 2: Create "glue" links with the smaller item **<u>off to the left</u>**.



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

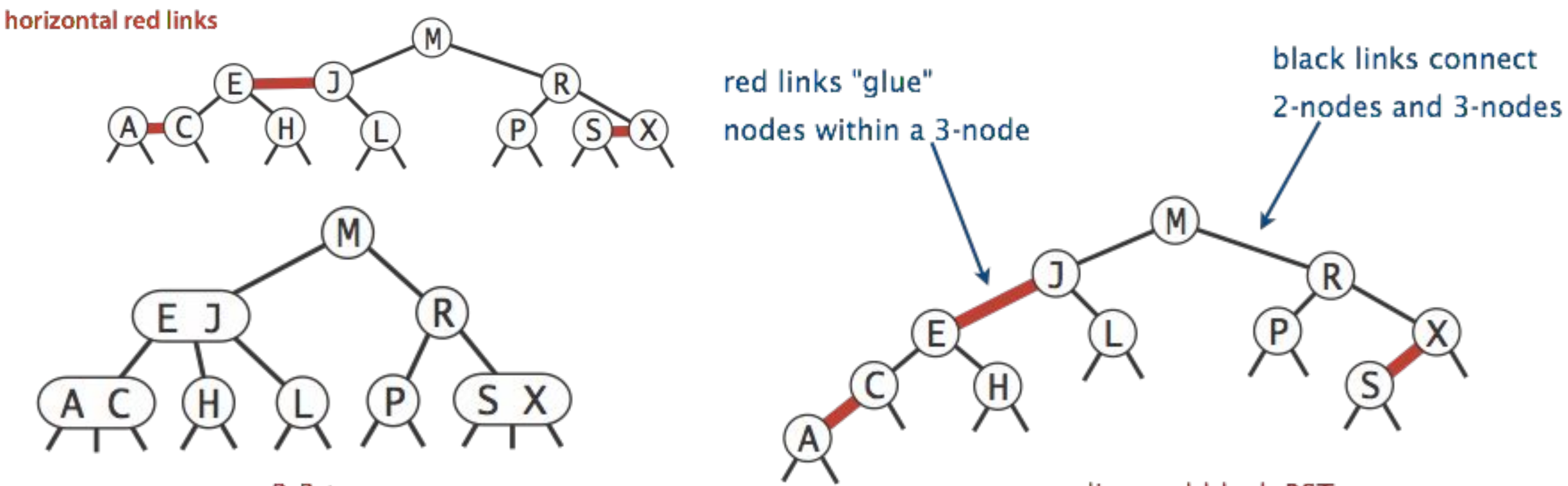


For convenience, we'll mark glue links as "**red**".

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.



2-3 tree

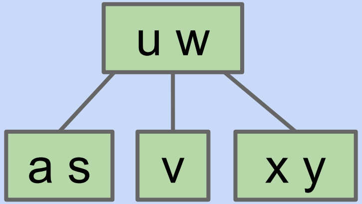
corresponding red-black BST

LLRB Properties

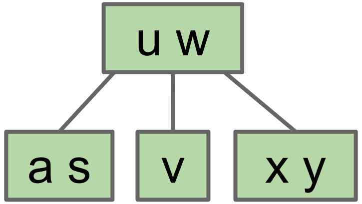
Lecture 18, CS61B, Spring 2024

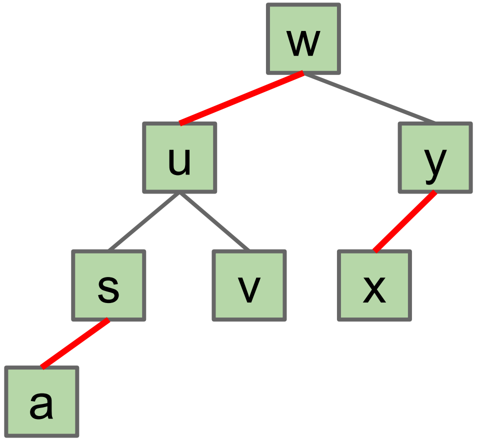
Left-Leaning Red Black Binary Search Tree (LLRB)

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



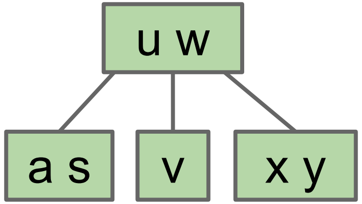
Draw the LLRB corresponding to the 2-3 tree shown below.





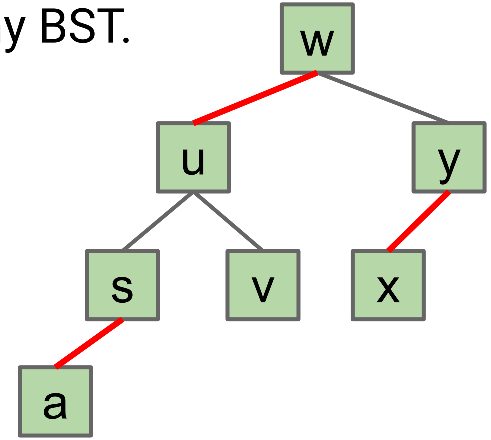
Left-Leaning Red Black Binary Search Tree (LLRB)

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



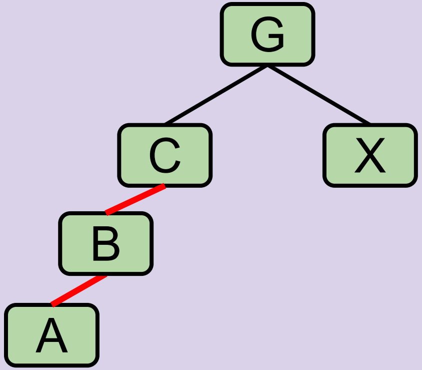
Searching an LLRB tree for a key is easy.

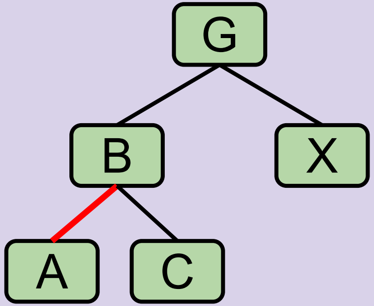
- Treat it exactly like any BST.

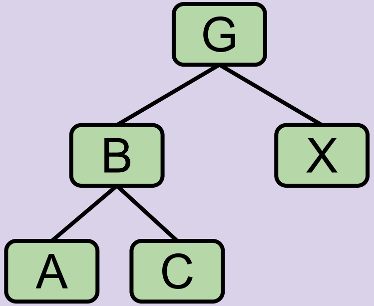


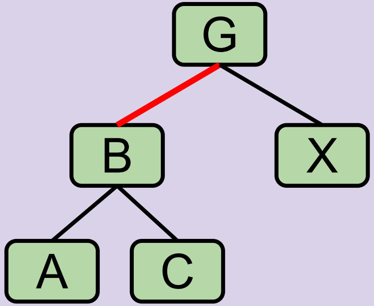
LLRB Problem #1: yellkey.com/view

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



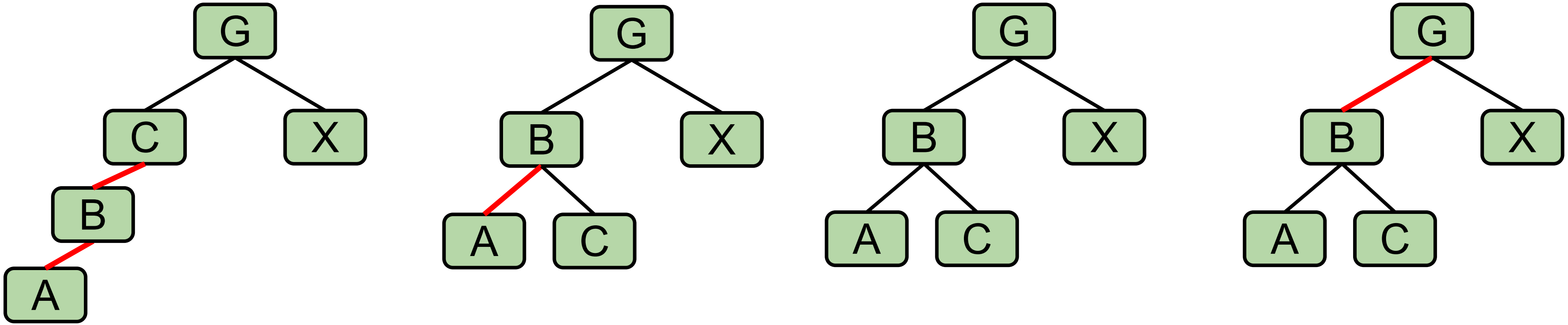




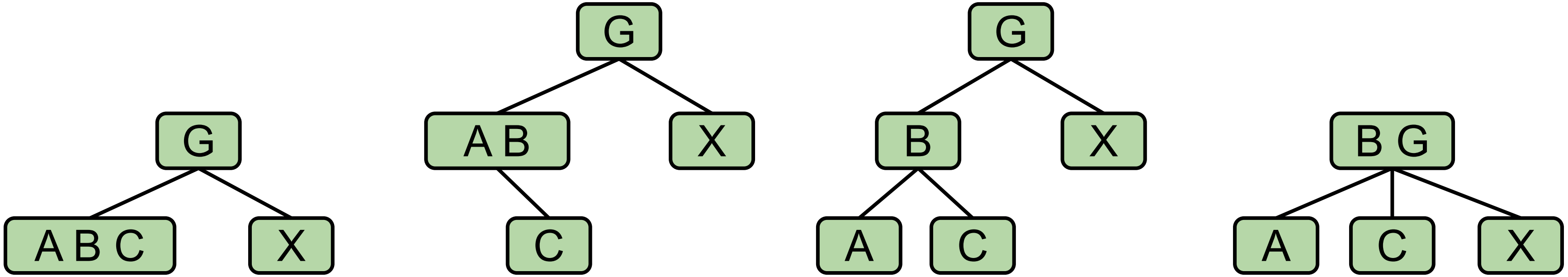


LLRB Problem #1: yellkey.com/view

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



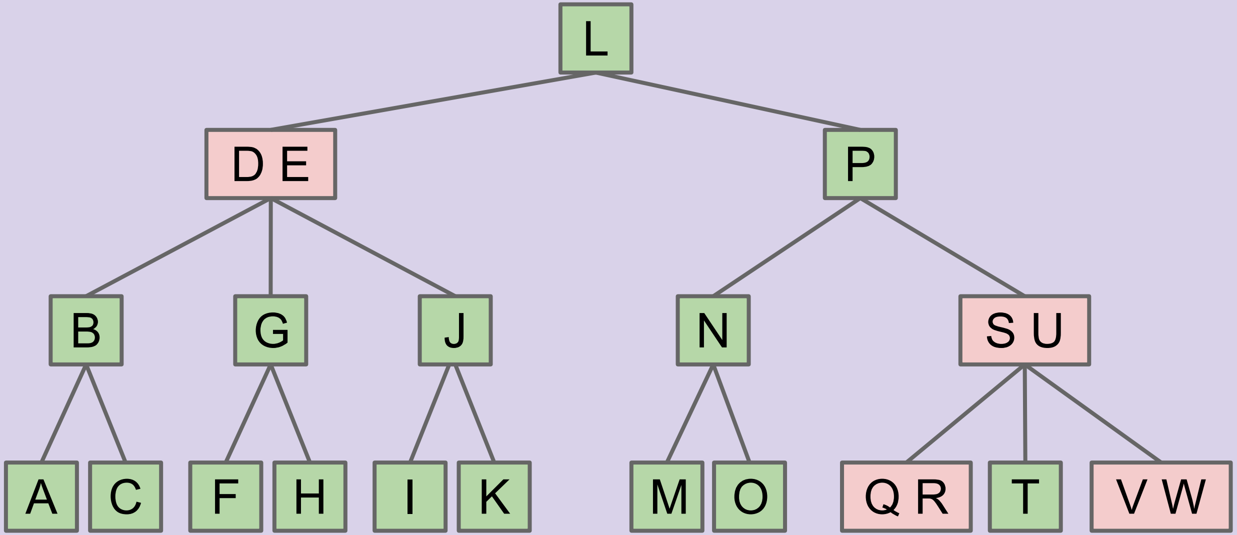
Equivalent 2-3



Invalid, has 4 node. Invalid, not balanced. Invalid, not balanced.

LLRB Problem #2: yellkey.com/chair

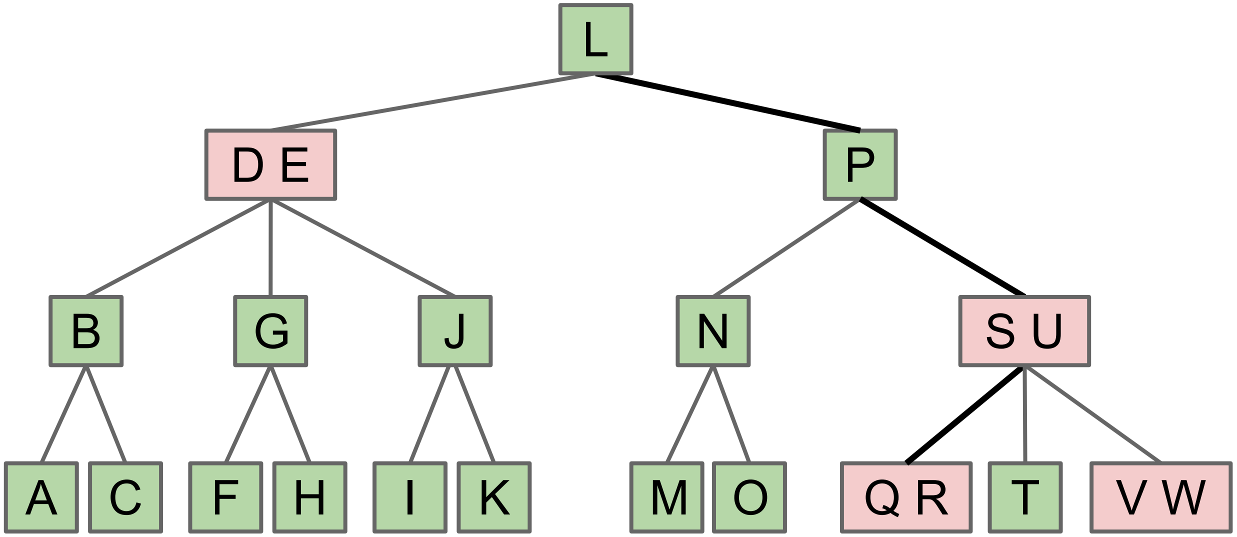
How tall is the corresponding LLRB for the 2-3 tree below? (3 - nodes in pink)



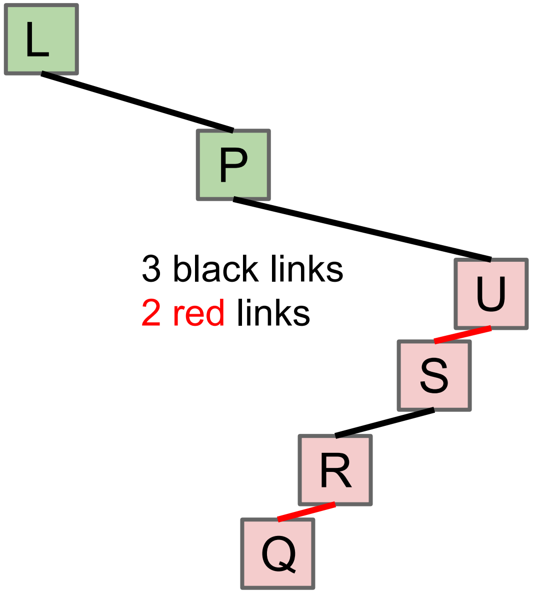
LLRB Problem #2: yellkey.com/chair

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 ~2x the height of its 2-3 tree.**

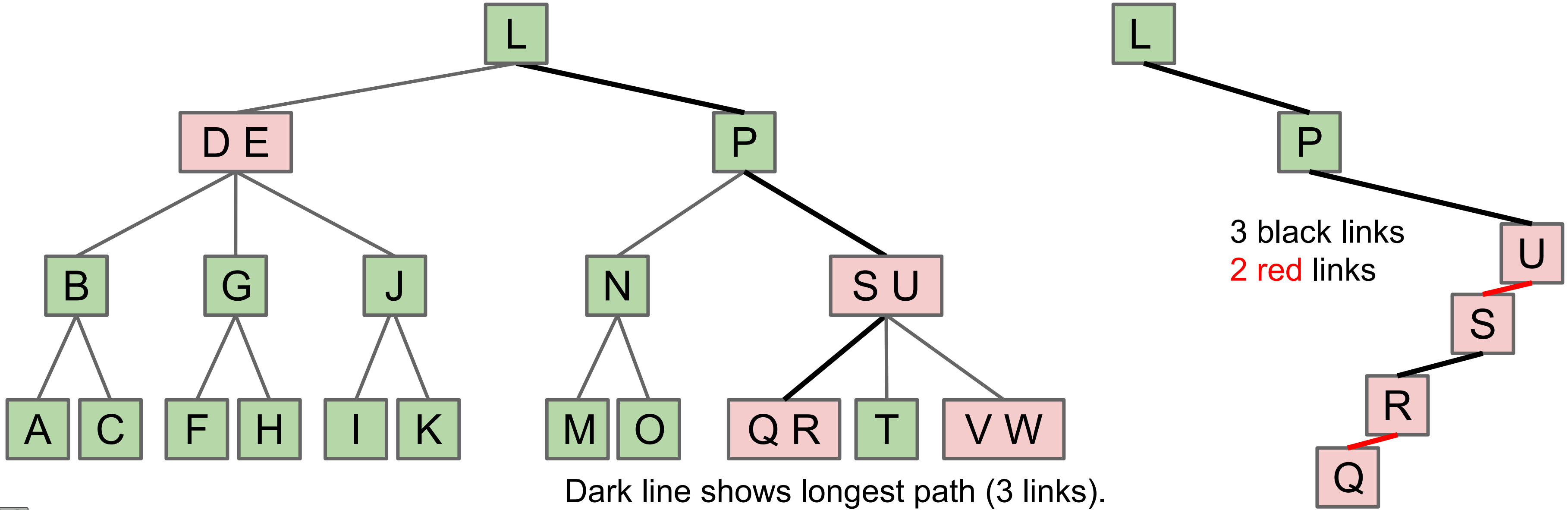


Dark line shows longest path (3 links).



LLRB Balance

Because 2-3 trees have logarithmic height, and the corresponding LLRB has height that is never more than ~2 times the 2-3 tree height, LLRBs also have logarithmic height!



Dark line shows longest path (3 links).

Extra: LLRB Invariants

Lecture 18, CS61B, Spring 2024

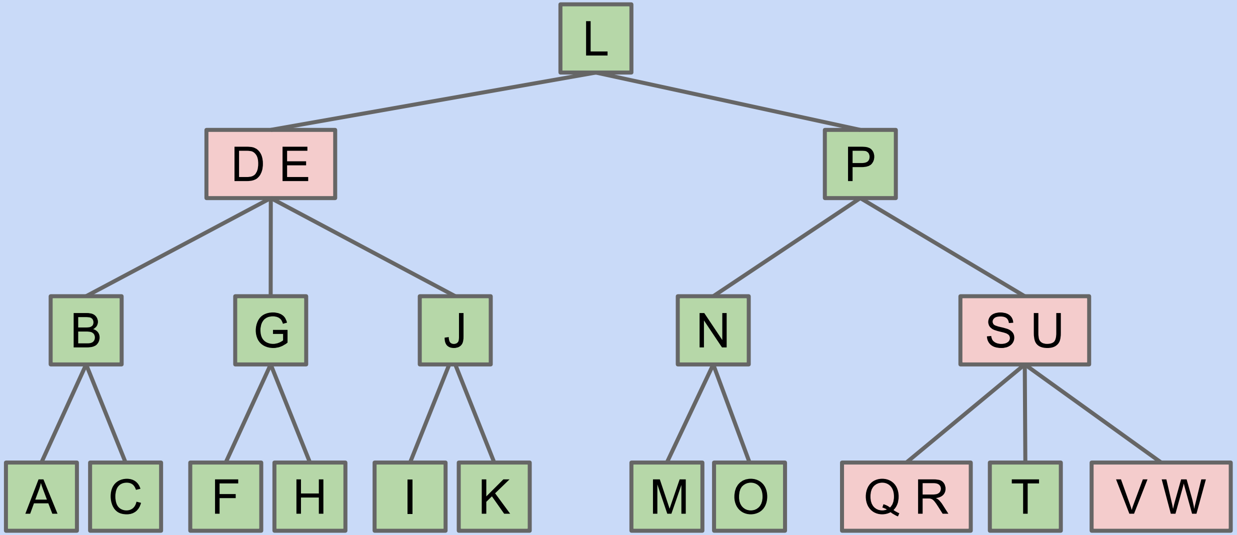
A somewhat more formal look at heights of LLRBs follows in the hidden slides after this one.

- This is covered in the web videos, but honestly, I don't think the argument is necessary.
- These slides include two invariants you might find interesting.

LLRB Height

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

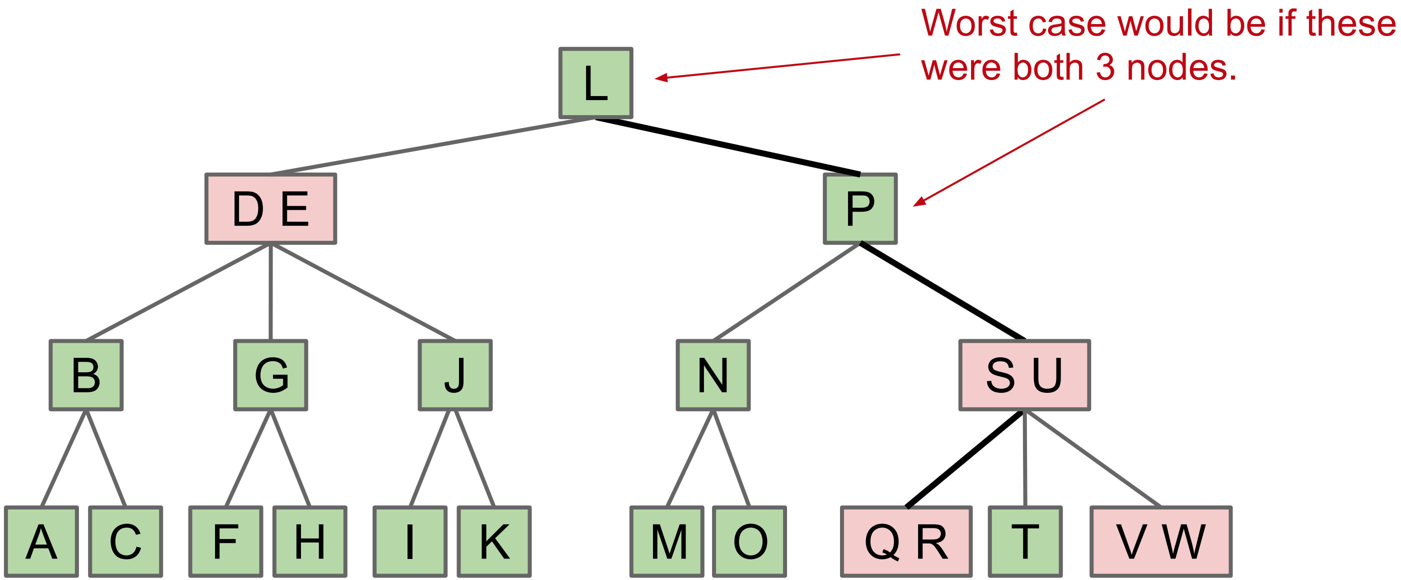
- What is the maximum height of the corresponding LLRB?



LLRB Height

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

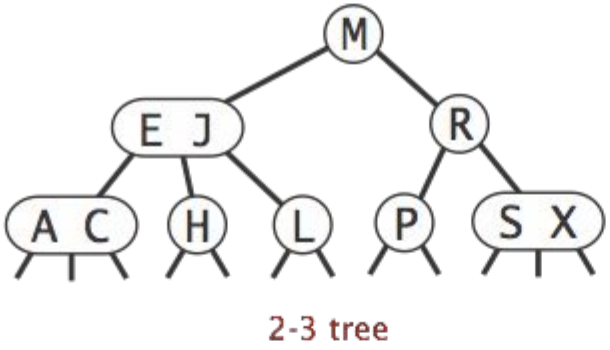
- What is the maximum height of the corresponding LLRB?
    - Total height is H (black) + H + 1 (red) = 2H + 1.

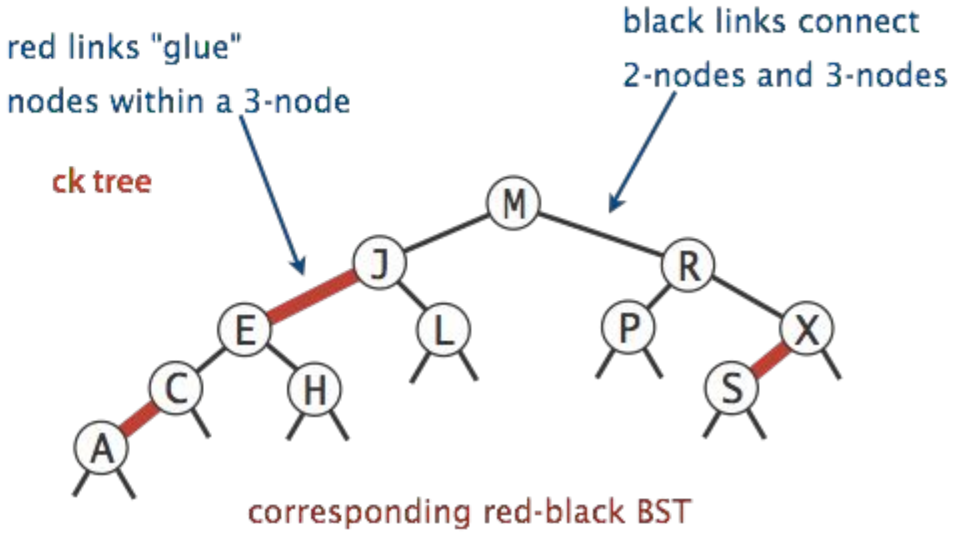


Left-Leaning Red Black Binary Search Tree (LLRB) Properties

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 null has same number of **black links** [because 2-3 trees have the same number of links to every leaf]. LLRBs are therefore balanced.





Extra Slide: Root-to-Leaf vs. Root-to-Null

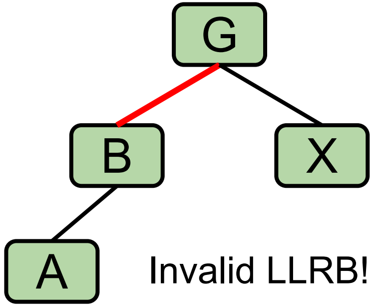
A version of this lecture from many years ago had a subtle error in its definition of "perfect black balance". Specifically, it stated:

- The number of black links to any leaf must be the same.

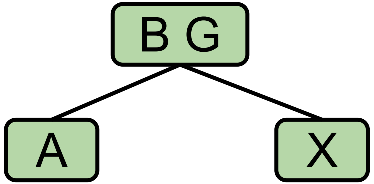
In fact, the correct invariant is:

- The number of black links to any null link must be the same.

An example of a red-black tree which satisfies the erroneous invariant, but has no corresponding 2-3 tree:



Invalid LLRB!

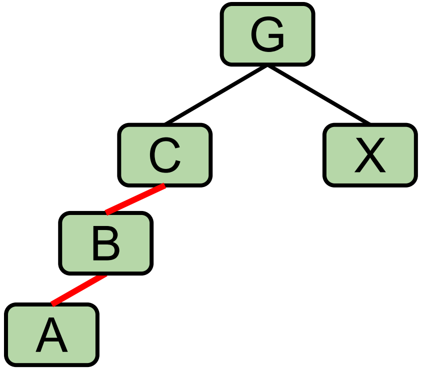


2-3 tree missing child.

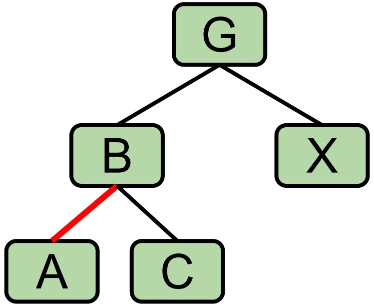
Left-Leaning Red Black Binary Search Tree (LLRB) Properties

Some handy LLRB properties:

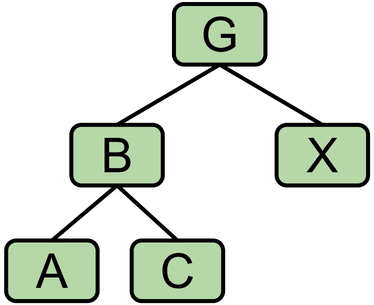
- No node has two red links [otherwise it'd be analogous to a 4 node, which are disallowed in 2-3 trees].
- Every path from root to null has same number of **black links** [because 2-3 trees have the same number of links to every leaf]. LLRBs are therefore balanced.



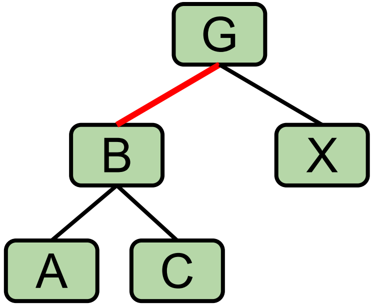
Invalid, B has two red links.



Invalid, not black balanced.



Invalid, not black balanced.



Valid

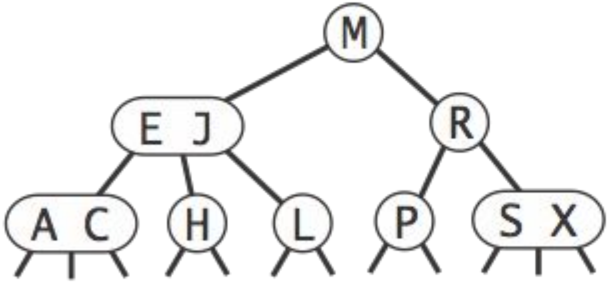
Maintaining Isometry with Rotations

Lecture 18, CS61B, Spring 2024

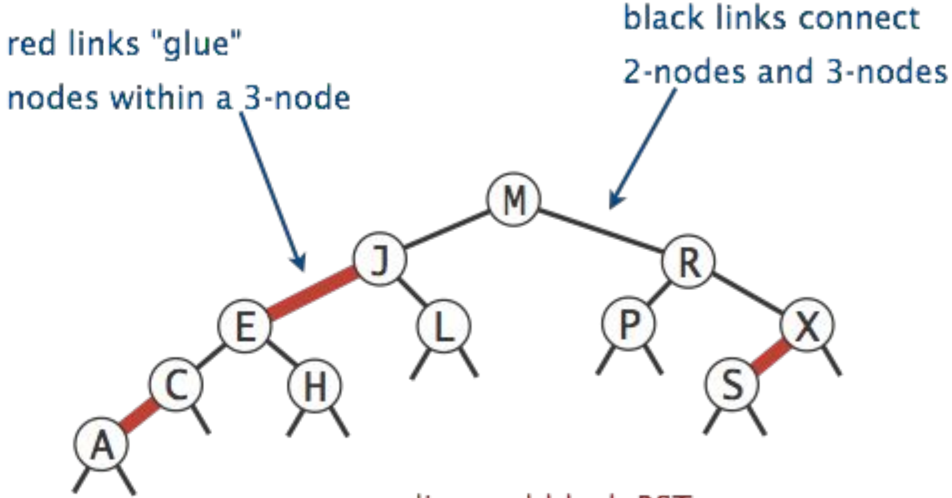
LLRB Construction

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.



2-3 tree



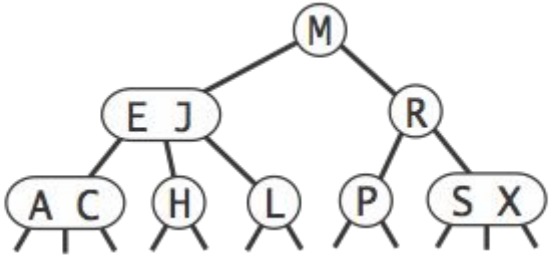
corresponding red-black BST

The 1-1 Mapping

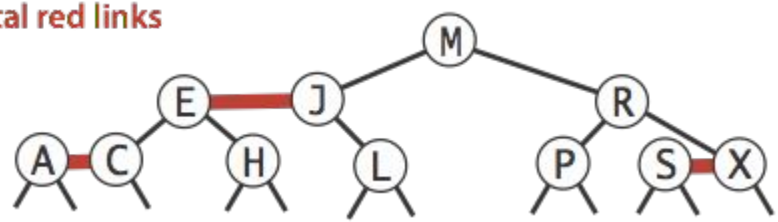
There exists a 1-1 mapping between:

- 2-3 Tree
- LLRB

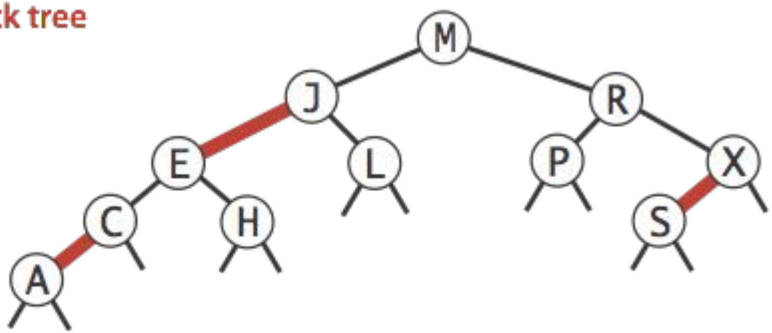
2-3 tree



horizontal red links



red–black tree

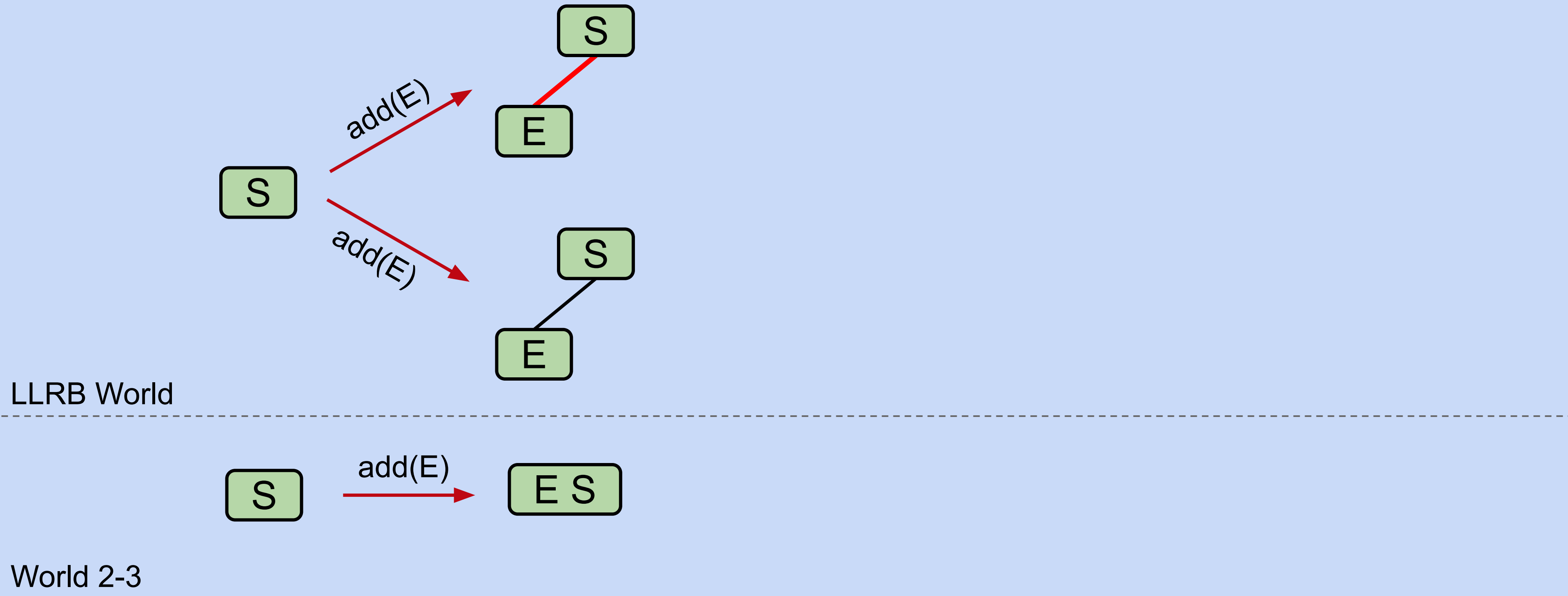


Implementation of an LLRB is based on maintaining this 1-1 correspondence.

- 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

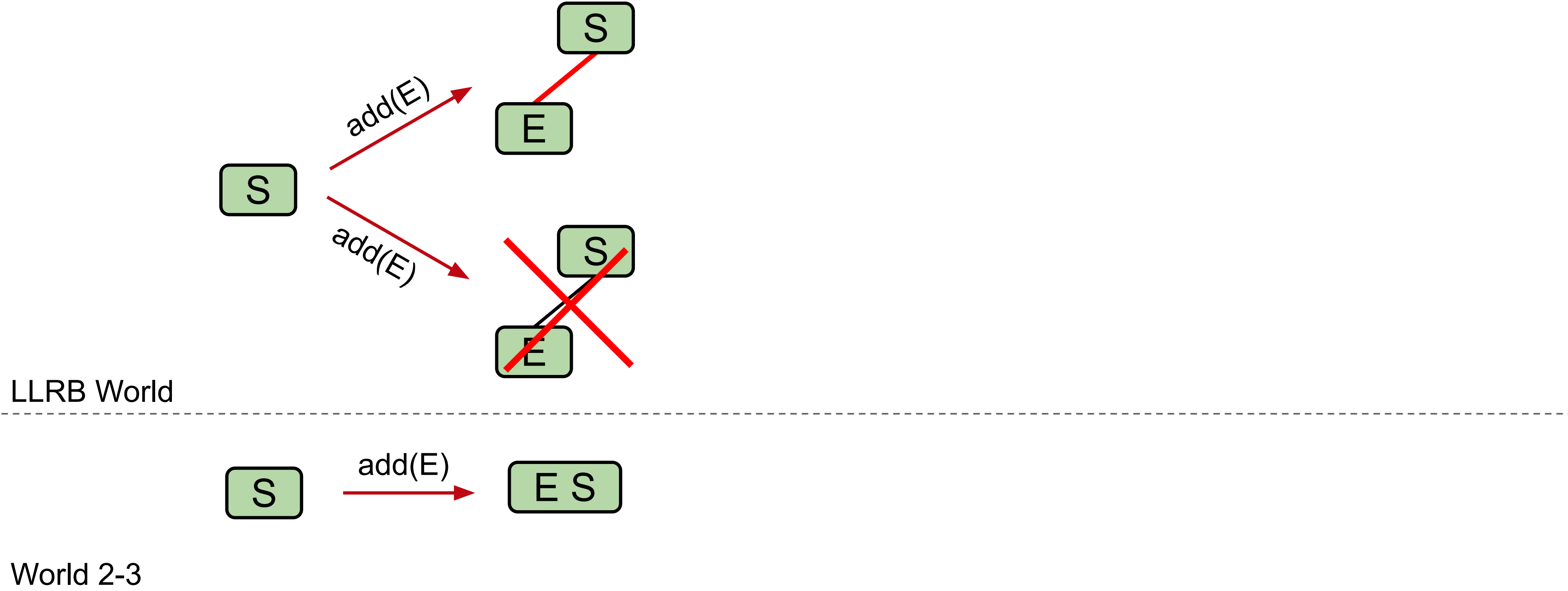
Should we use a red or black link when inserting?



Design Task #1: Insertion Color

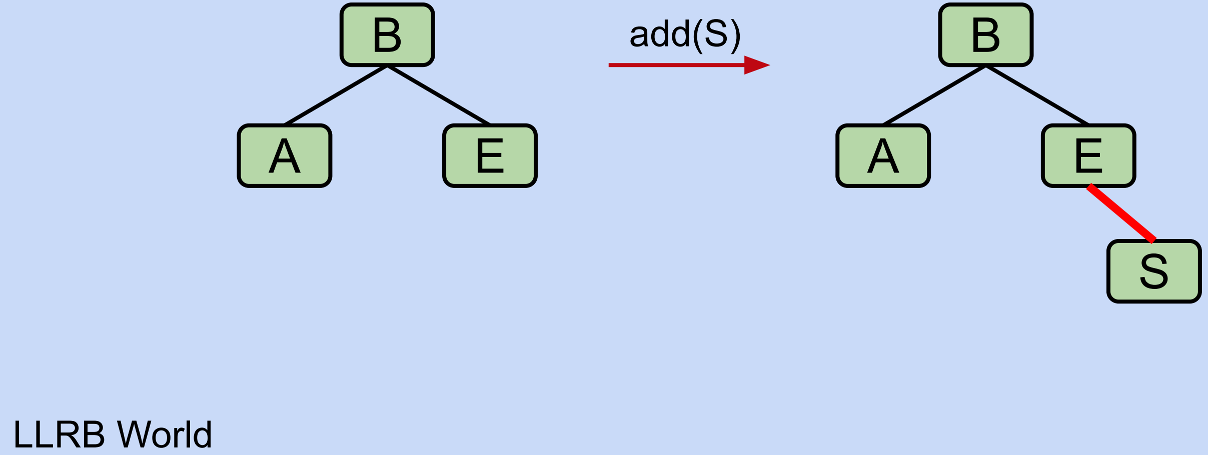
Should we use a red or black link when inserting?

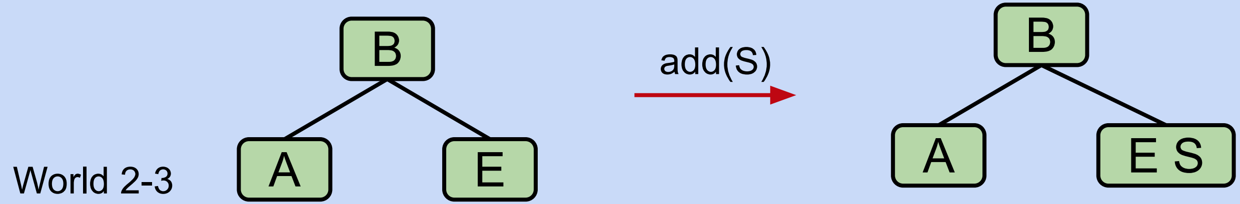
- Use red! In 2-3 trees new values are ALWAYS added to a leaf node (at first).



Design Task #2: Insertion on the Right

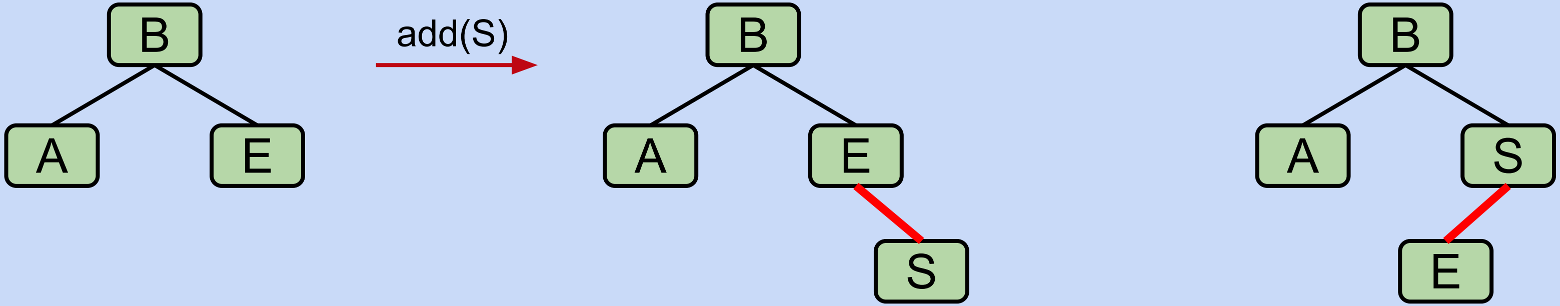
Suppose we have leaf E, and insert S with a red link. What is the problem below, and what do we do about it?





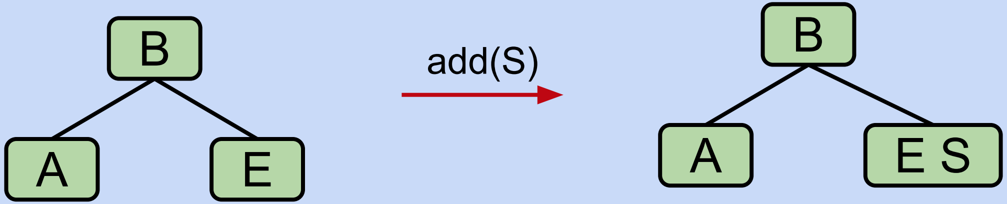
Design Task #2: Insertion on the Right

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. What rotation fixes this?



Hint: This is the correct representation of the 2-3 tree.

LLRB World

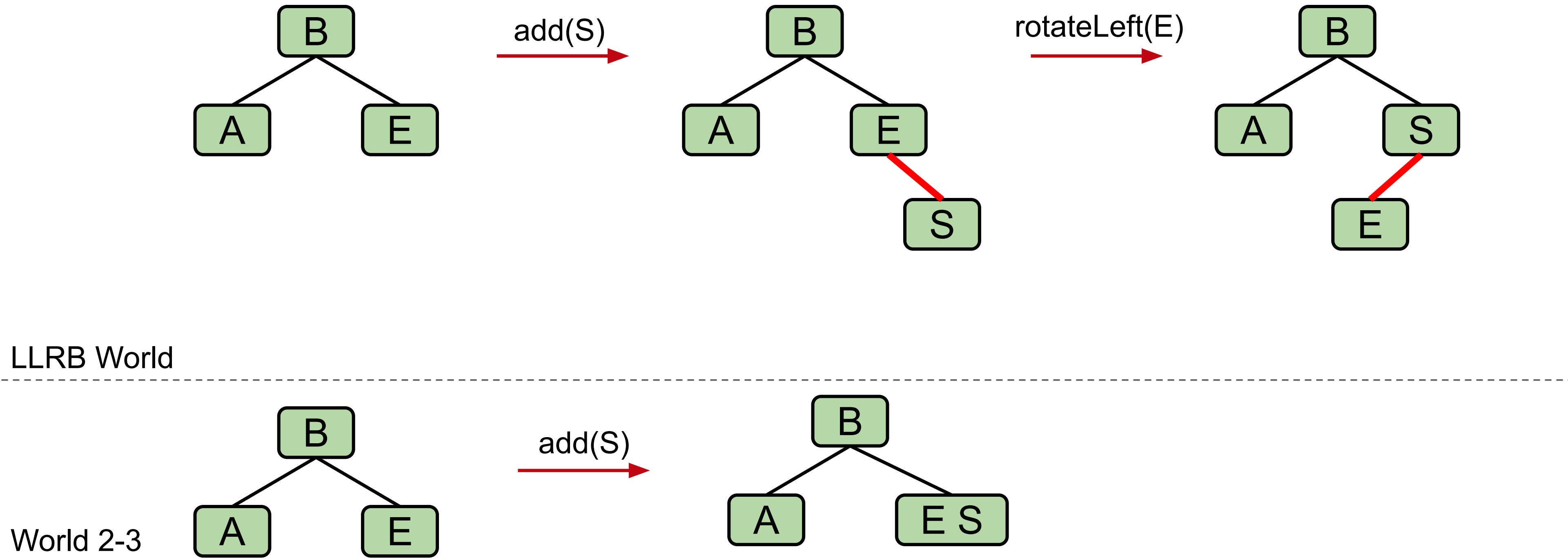


What rotation operation gives us this tree?

World 2-3

Design Task #2: Insertion on the Right

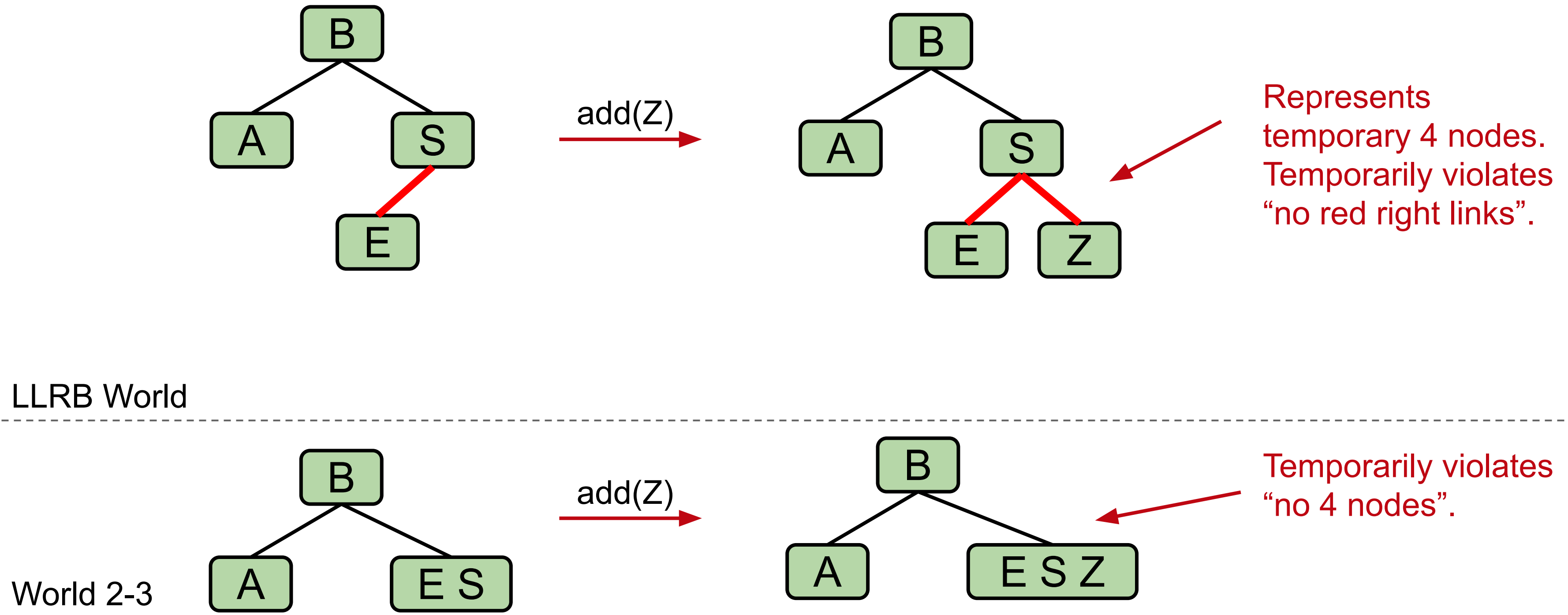
Suppose we have leaf E, and insert S with a red link. What is the problem below, and what do we do about it: Right links aren't allowed, so rotateLeft(E).



New Rule: Representation of Temporary 4-Nodes

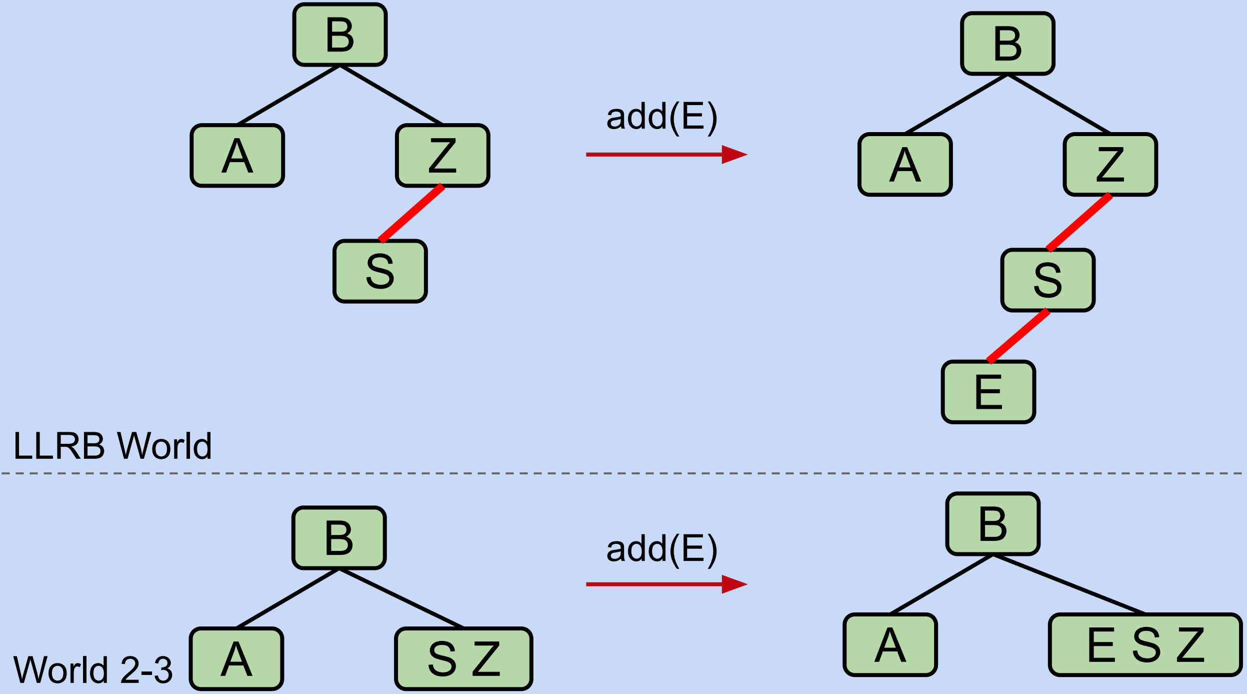
We will represent temporary 4-nodes as BST nodes with two red links.

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



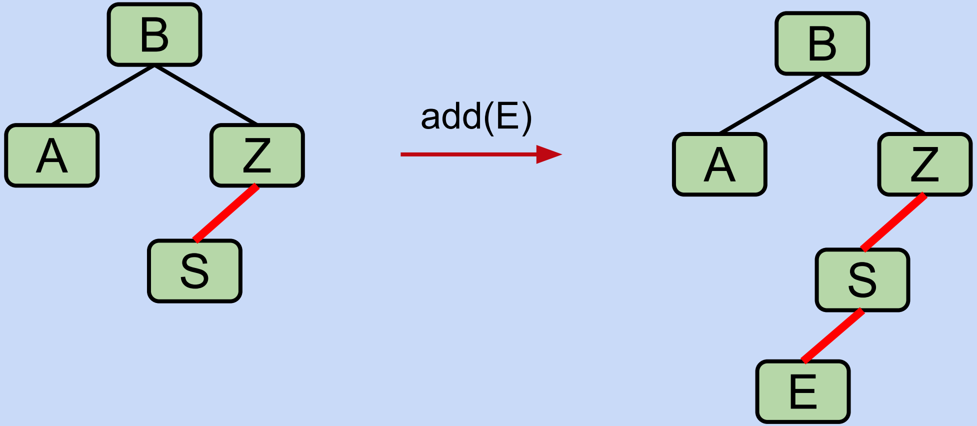
Design Task #3: Double Insertion on the Left

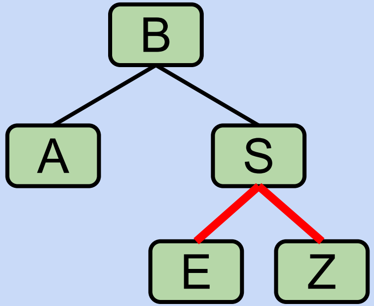
Suppose we have the LLRB below and insert E. We end up with the wrong representation for our temporary 4 node. What should we do so that the temporary 4 node has 2 red children (one left, one right) as expected?



Design Task #3: Double Insertion on the Left

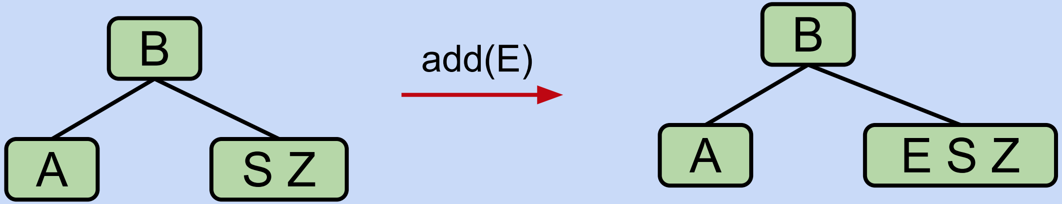
Suppose we have the LLRB below and insert E. We end up with the wrong representation for our temporary 4 node. What should we do so that the temporary 4 node has 2 red children (one left, one right) as expected?





Hint: This is the correct representation of the 2-3 tree.

LLRB World



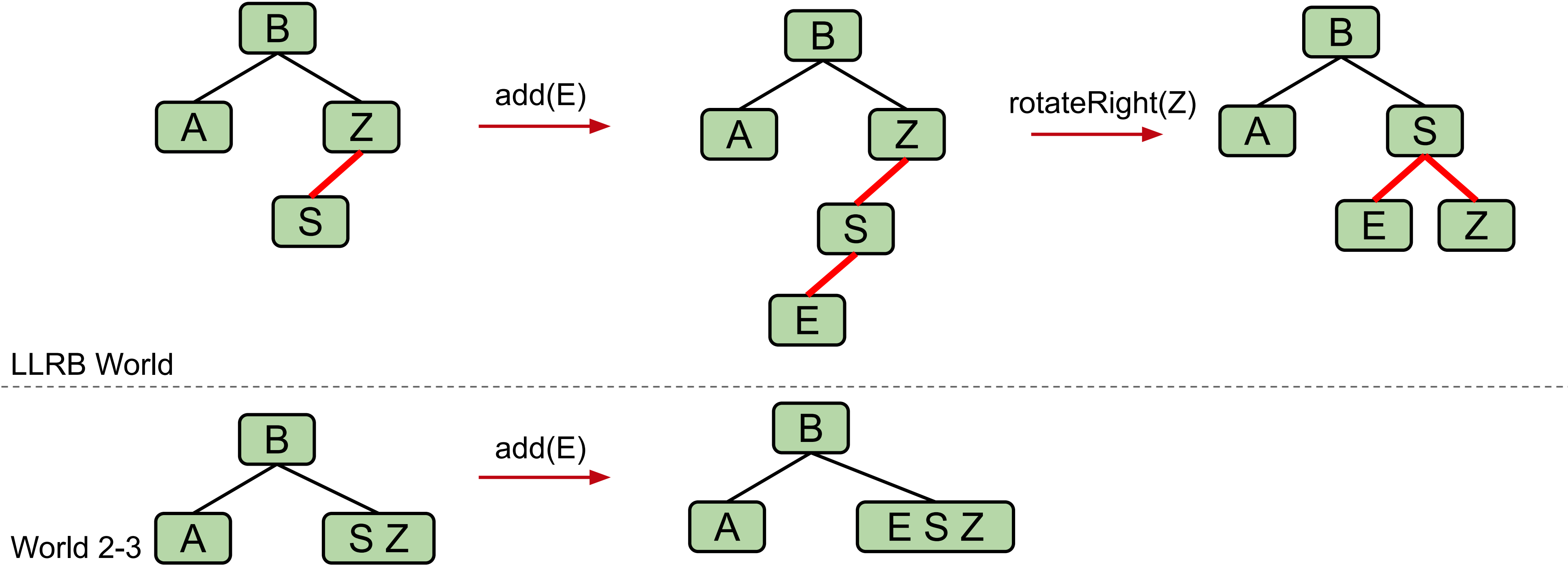
World 2-3

What rotation operation gives us this tree?

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?

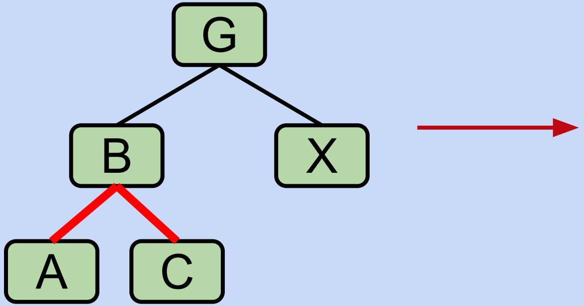
- Rotate Z right.



Design Task #4: Splitting Temporary 4-Nodes

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

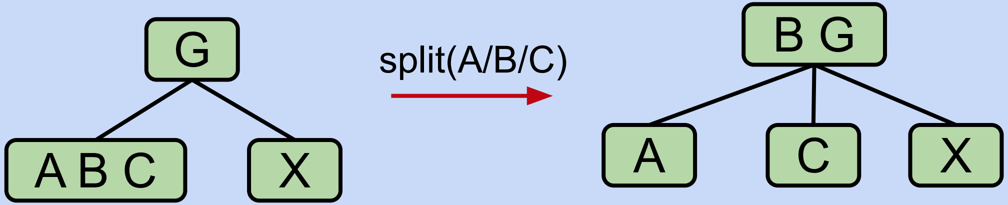
- Try to figure this one out! It's a particularly interesting puzzle.



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

LLRB World

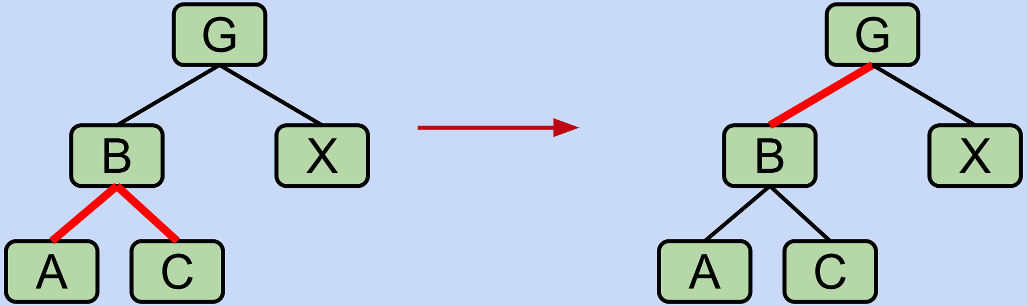
World 2-3



Design Task #4: Splitting Temporary 4-Nodes

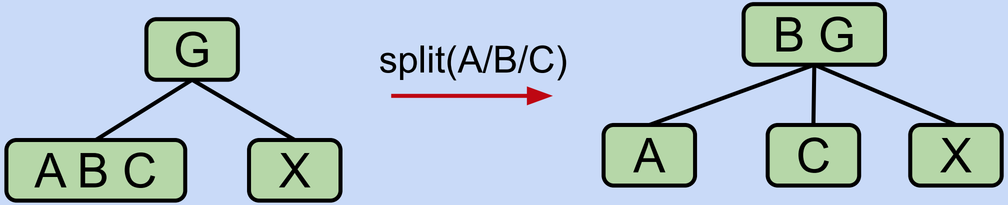
Suppose we have the LLRB below which includes a temporary 4 node. What should we do next?

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



Hint 2: This is the correct representation of the 2-3 tree.

LLRB World



How do we get this tree?

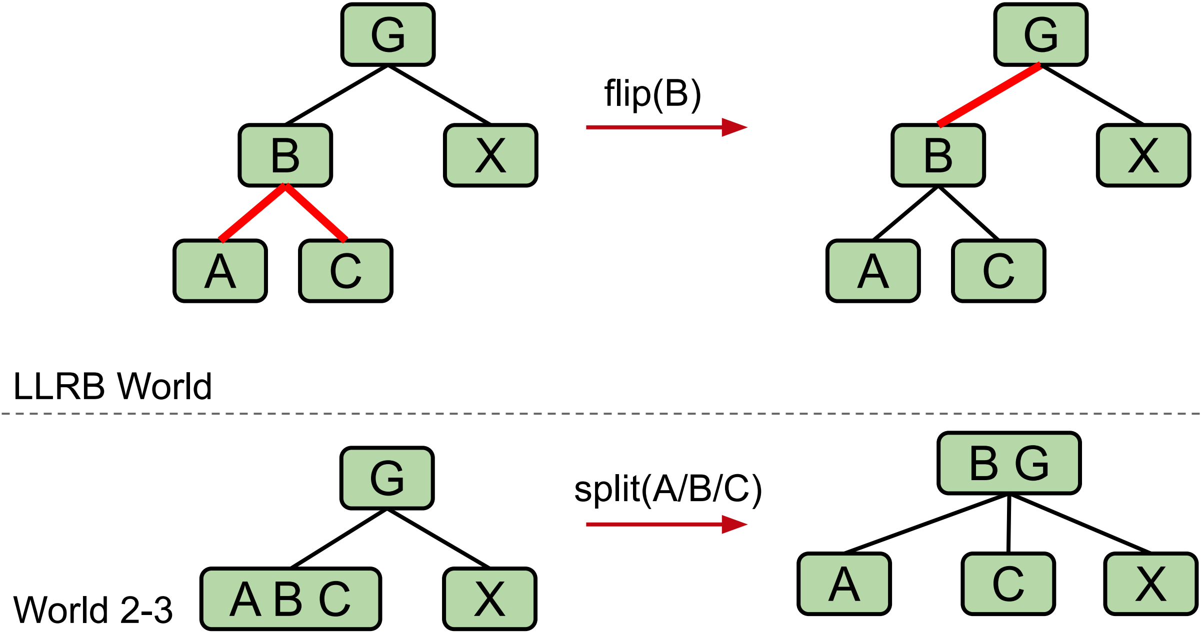
Hint 3: We don't need rotation.

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.

… 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.

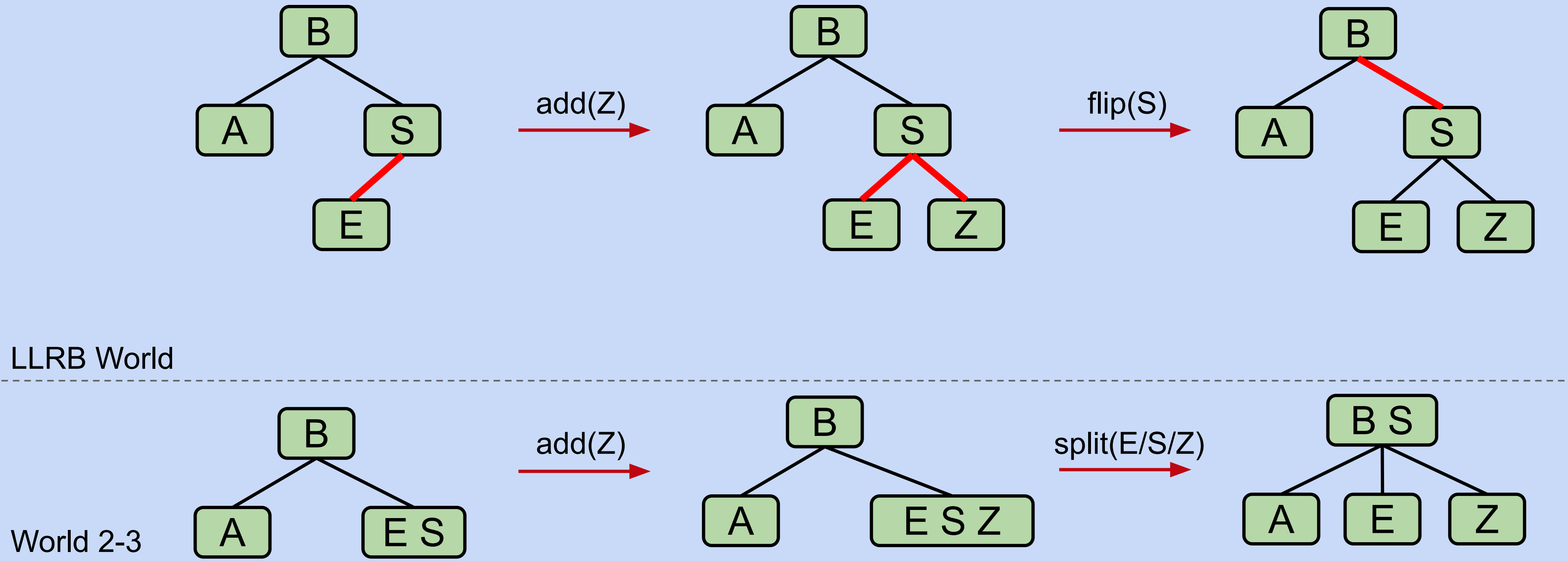
One last detail: Cascading operations.

- 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.

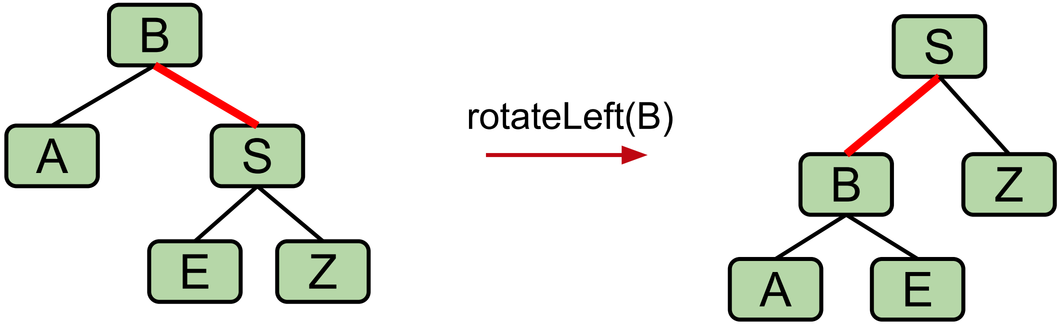
- Color flip yields an invalid tree. Why? What’s next?



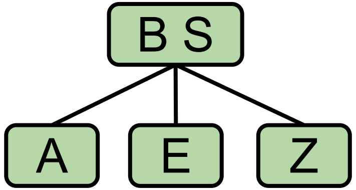
Cascading Balance Example

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

Lecture 18, CS61B, Spring 2024

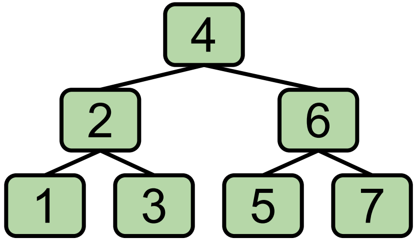
Insertion of 7 through 1

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.

- Or see this video walkthrough of solution.



Runtime and Implementation

Lecture 18, CS61B, Spring 2024

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).

Search Tree Summary

Lecture 18, CS61B, Spring 2024

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.