CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 29: Red-Black Trees

Announcements

- Midterm 2 on Friday
- Wanna be a 250 SA? More information coming soon, watch Piazza
- **NO RECITATION THIS WEEK**

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

What is the runtime in terms of n*?* $O(n)$

$$\log(n) \leq d \leq n$$

AVL Summary

- We want shallow BSTs (it makes **`find`**, **`insert`**, **`remove`** faster)

AVL Summary

- We want shallow BSTs (it makes **`find`**, **`insert`**, **`remove`** faster)
- Enforcing AVL constraints makes our BSTs shallow
  - The constraints are |height(right) - height(left)| ≤ 1
  - It will guarantee $d = O(\log(n))$

AVL Summary

- We want shallow BSTs (it makes **`find`**, **`insert`**, **`remove`** faster)
- Enforcing AVL constraints makes our BSTs shallow
  - The constraints are |height(right) - height(left)| ≤ 1
  - It will guarantee $d = O(\log(n))$
- Adding/removing from a BST changes height by at most 1
- A rotation can also change a BST height by at most 1

AVL Summary

- We want shallow BSTs (it makes **`find`**, **`insert`**, **`remove`** faster)
- Enforcing AVL constraints makes our BSTs shallow
  - The constraints are |height(right) - height(left)| ≤ 1
  - It will guarantee $d = O(\log(n))$
- Adding/removing from a BST changes height by at most 1
- A rotation can also change a BST height by at most 1
- Therefore after **`insert`**/**`remove`** into an AVL tree, we can reinforce AVL constraints with one (or two) rotations
  - We only need to make one trip back up the tree to do so
  - Therefore **`insert`**/**`remove`** is still $O(d) = O(\log(n))$

AVL Tree

What was our initial goal?

AVL Tree

What was our initial goal? **To constrain the depth of the tree**

AVL Tree

What was our initial goal? **To constrain the depth of the tree**

How did we accomplish it?

AVL Tree

What was our initial goal? **To constrain the depth of the tree**

How did we accomplish it? **By keeping the tree balanced (subtree heights within 1 of each other)**

AVL Tree

What was our initial goal? **To constrain the depth of the tree**

How did we accomplish it? **By keeping the tree balanced (subtree heights within 1 of each other)**

This approach is indirect, and a bit more restrictive than it has to be

Maintaining Balance - Another Approach

Enforcing height-balance is too strict (May do “unnecessary” rotations)

Weaker (and more direct) restriction:

- Balance the depth of empty tree nodes
- If ***a***, ***b*** are EmptyTree nodes, then enforce that for all ***a***, ***b***:
  - depth(***b***) ≥ depth(***a***) ≥ (depth(***b***) ÷ 2)

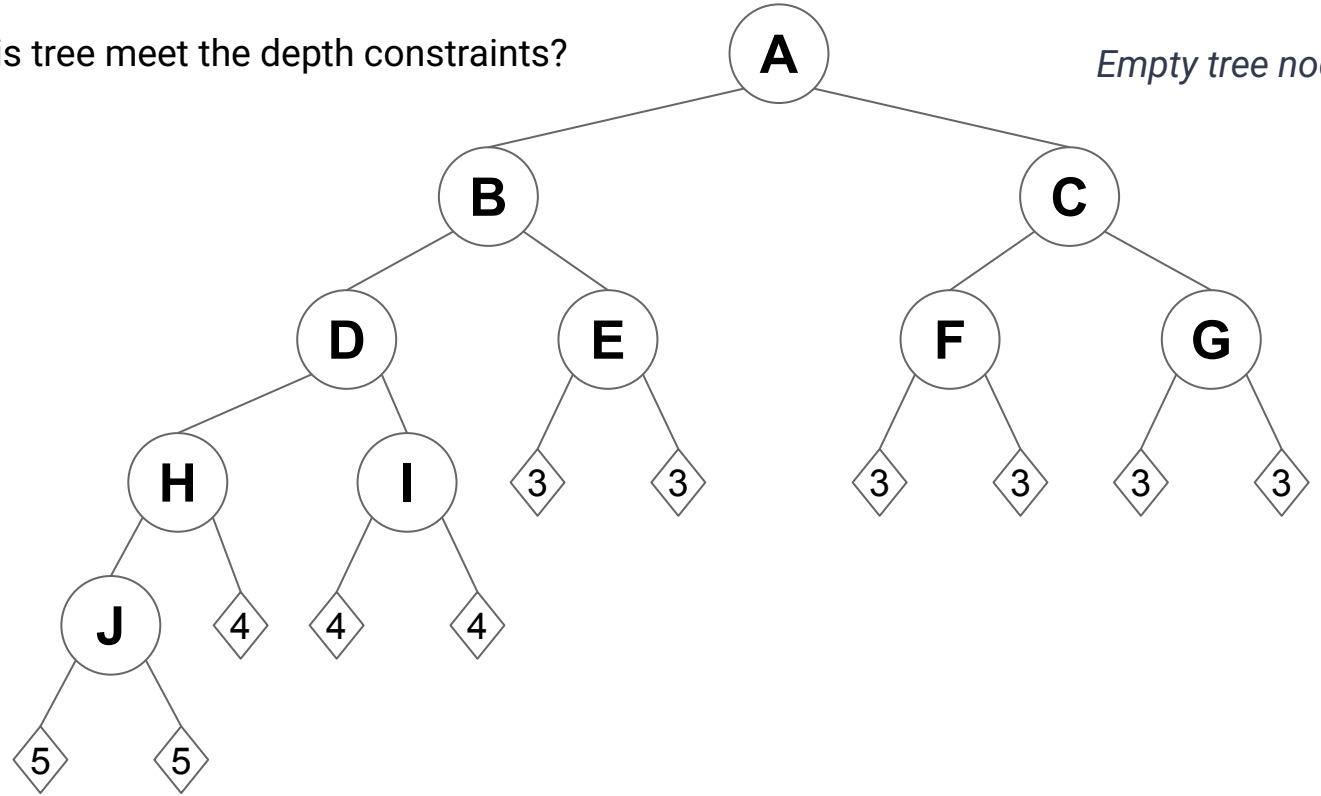
    or

  - depth(***a***) ≥ depth(***b***) ≥ (depth(***a***) ÷ 2)

Like with all BST properties we've discussed, this also has to hold true for ALL subtrees

Depth Balancing

Does this tree meet the depth constraints?



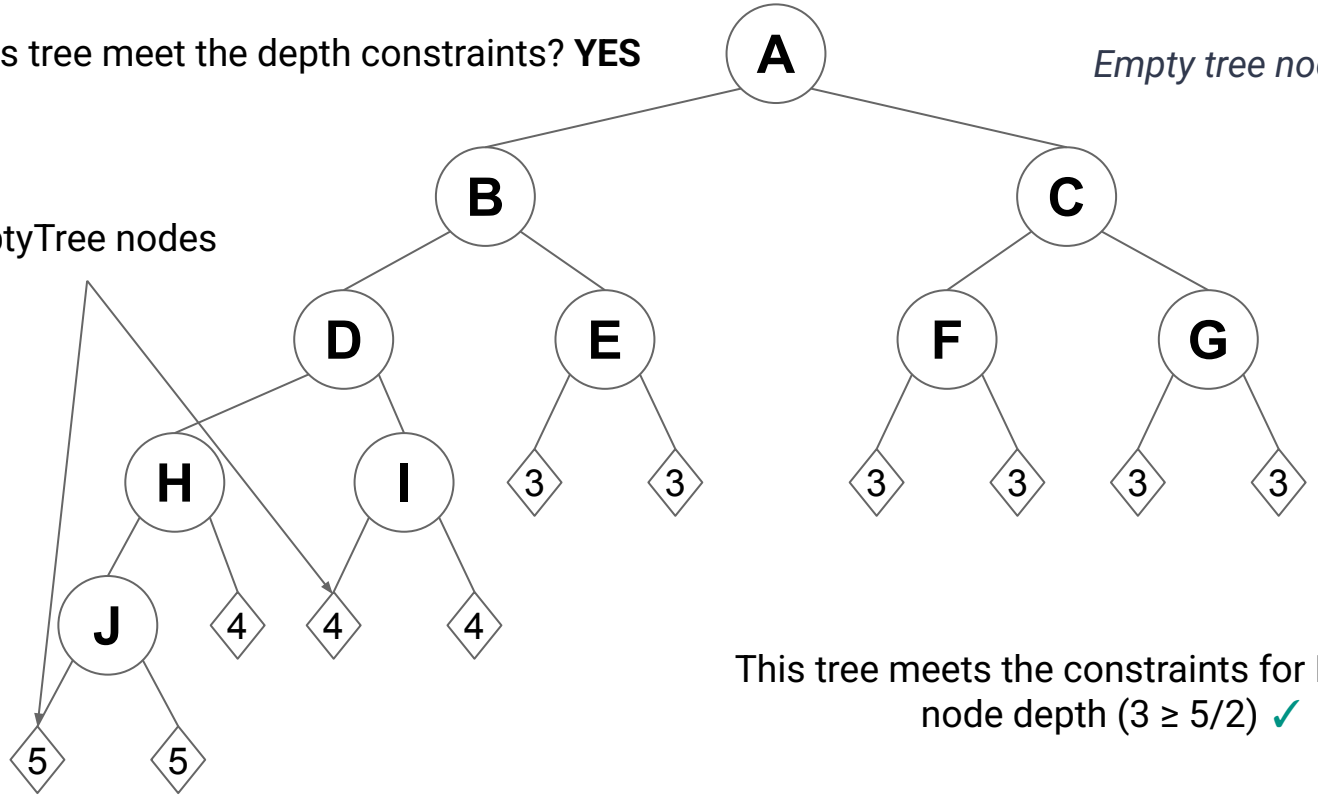
Empty tree nodes are labeled with their depth in this diagram

Depth Balancing

Does this tree meet the depth constraints? **YES**

Empty tree nodes are labeled with their depth in this diagram

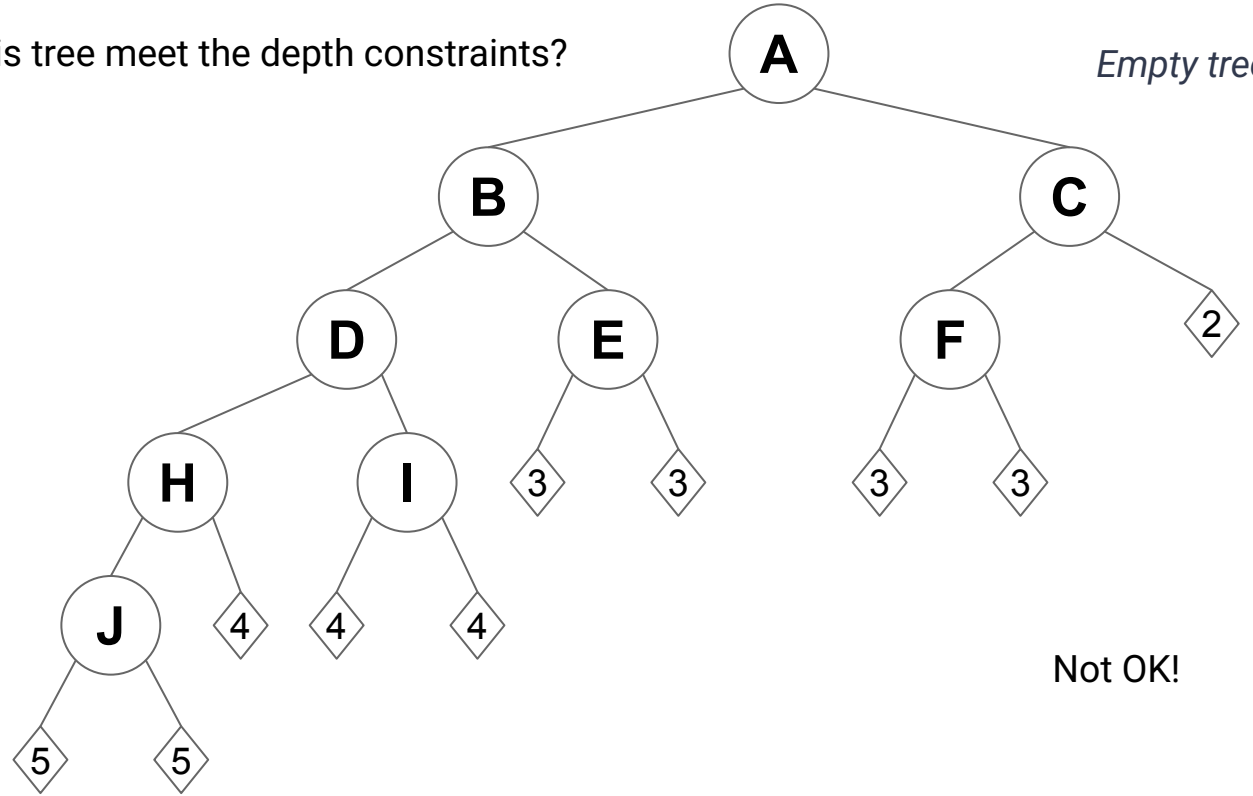
EmptyTree nodes



This tree meets the constraints for EmptyTree node depth (3 ≥ 5/2) ✓

Depth Balancing

Does this tree meet the depth constraints?

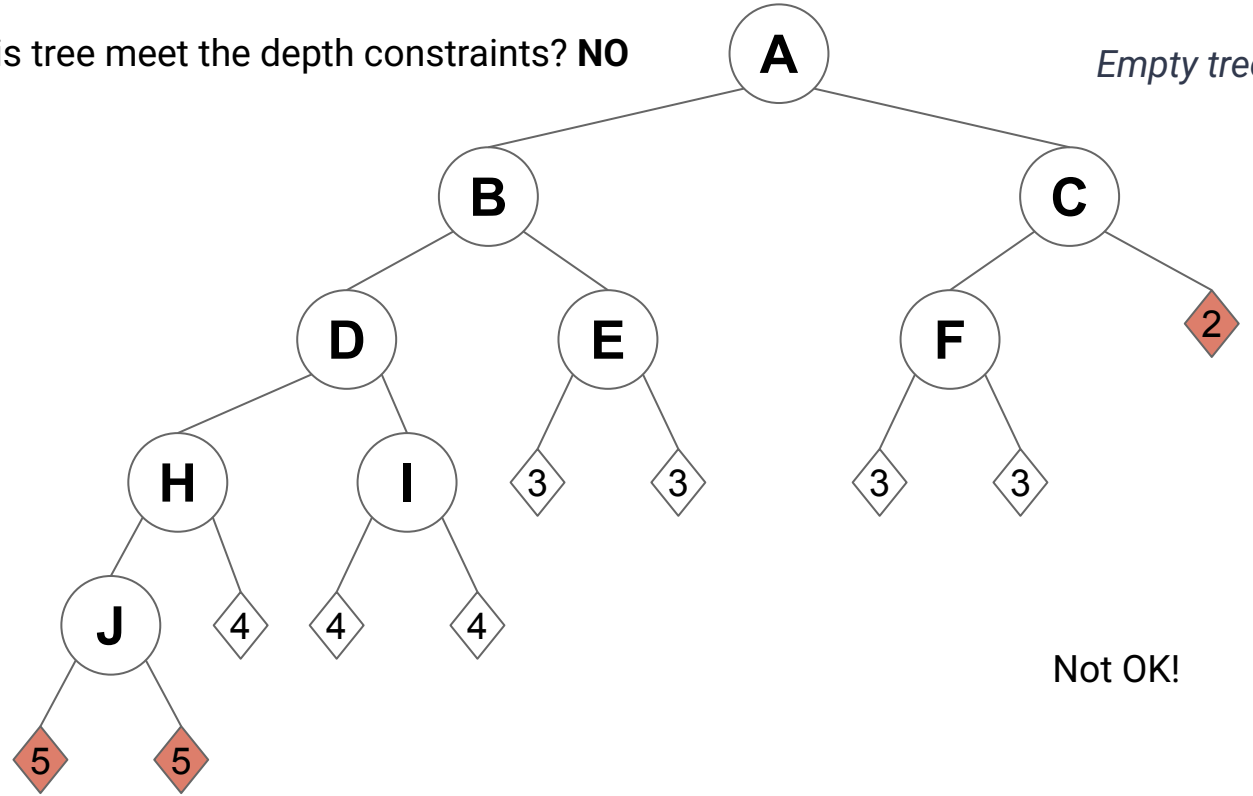


Empty tree nodes are labeled with their depth in this diagram

Not OK!

Depth Balancing

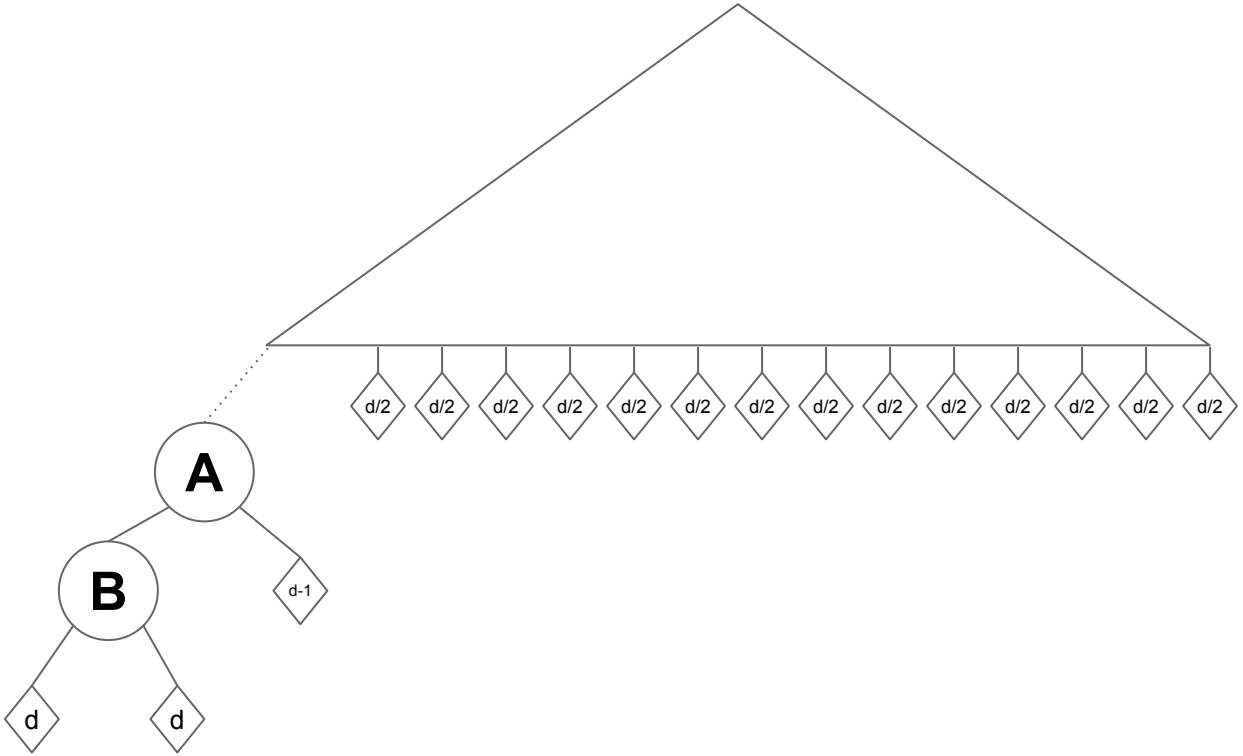
Does this tree meet the depth constraints? **NO**



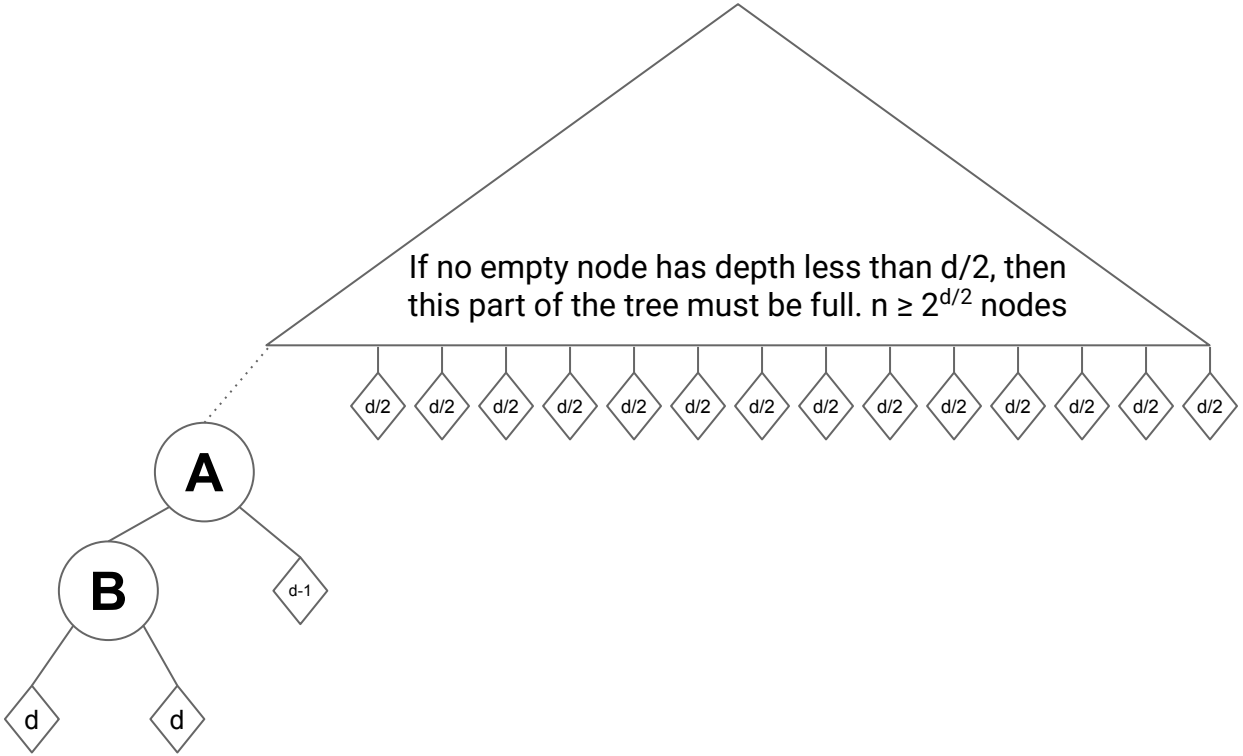
Empty tree nodes are labeled with their depth in this diagram

Not OK!

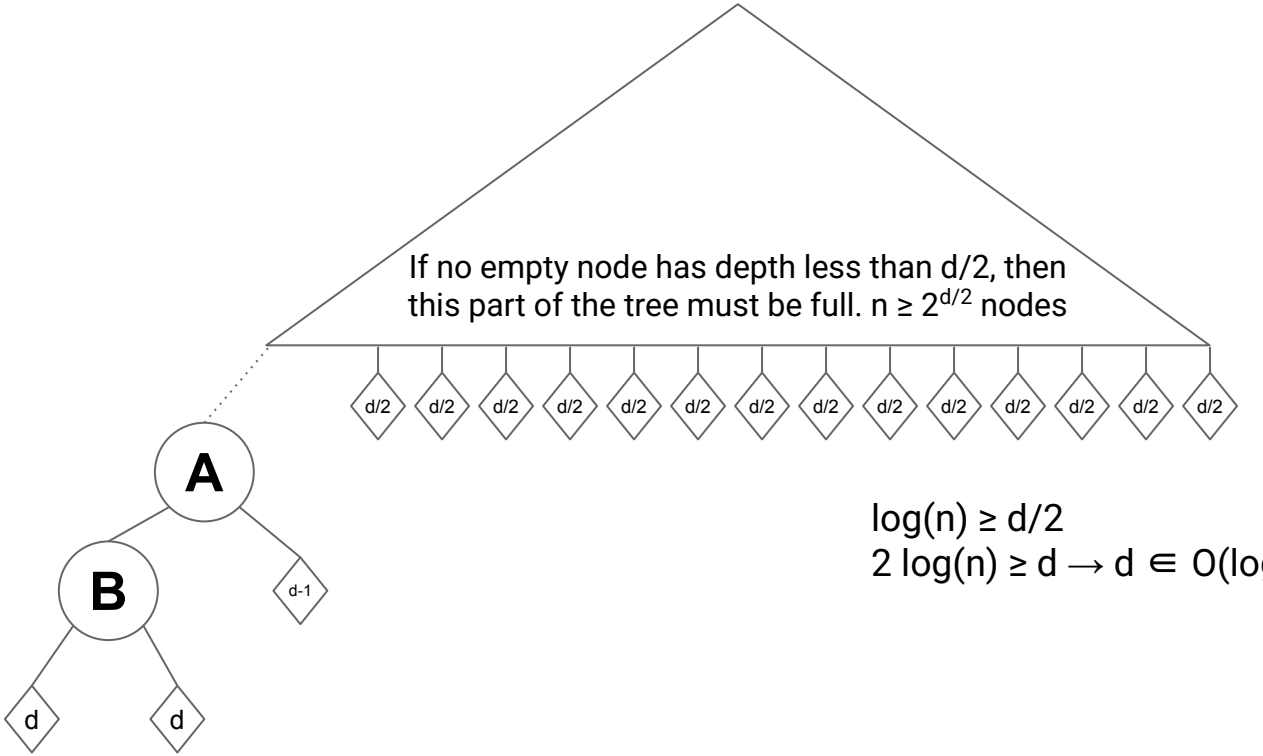
Depth Balancing



Depth Balancing



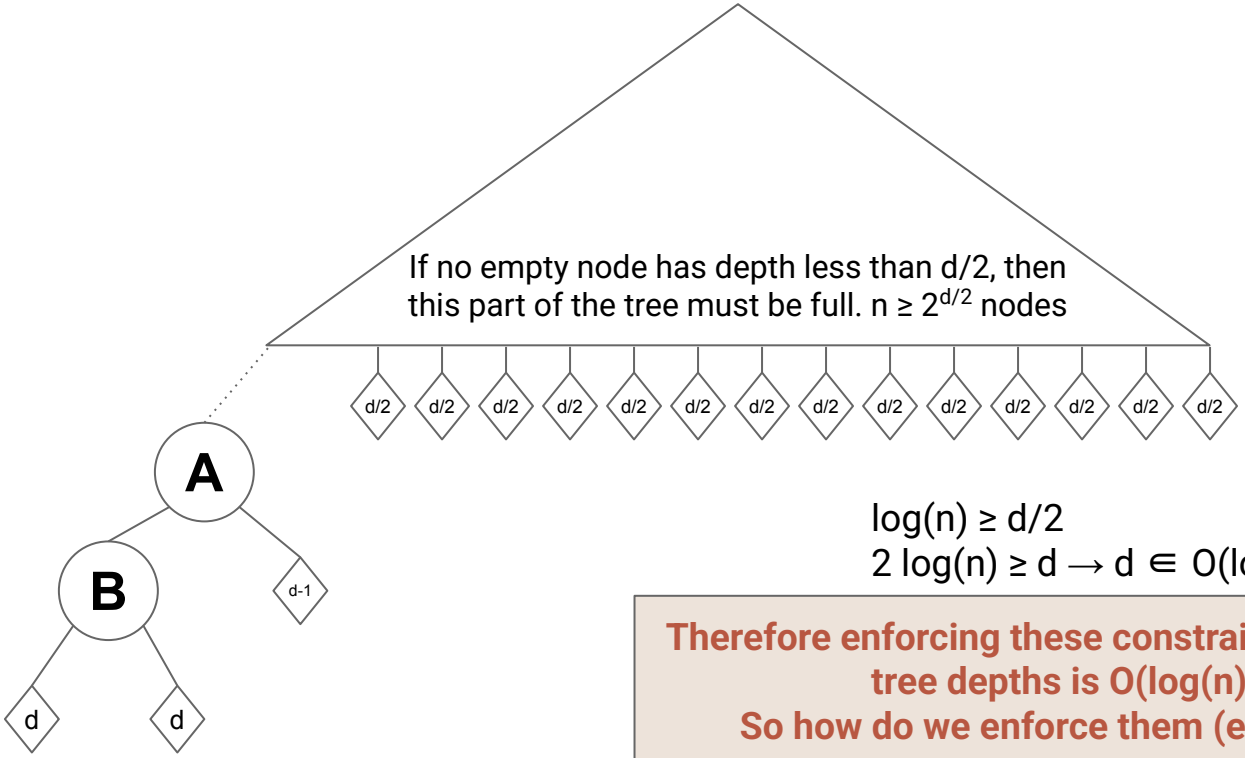
Depth Balancing



$\log(n) \geq d/2$

$2 \log(n) \geq d \rightarrow d \in O(\log(n))$

Depth Balancing



$\log(n) \geq d/2$

$2 \log(n) \geq d \rightarrow d \in O(\log(n))$

Therefore enforcing these constraints means that tree depths is O(log(n))...
So how do we enforce them (efficiently)?

Red-Black Trees

To Enforce the Depth Constraint on empty nodes:

1. Color each node red or black
    a. The # of black nodes from each empty node to root must be same
    b. The parent of a red node must always be black
2. On insertion (or deletion)
    a. Inserted nodes are red (won't break 1a)
    b. Repair violations of 1b by rotating and/or recoloring
        i. Make sure repairs don't break 1a

Red-Black Trees

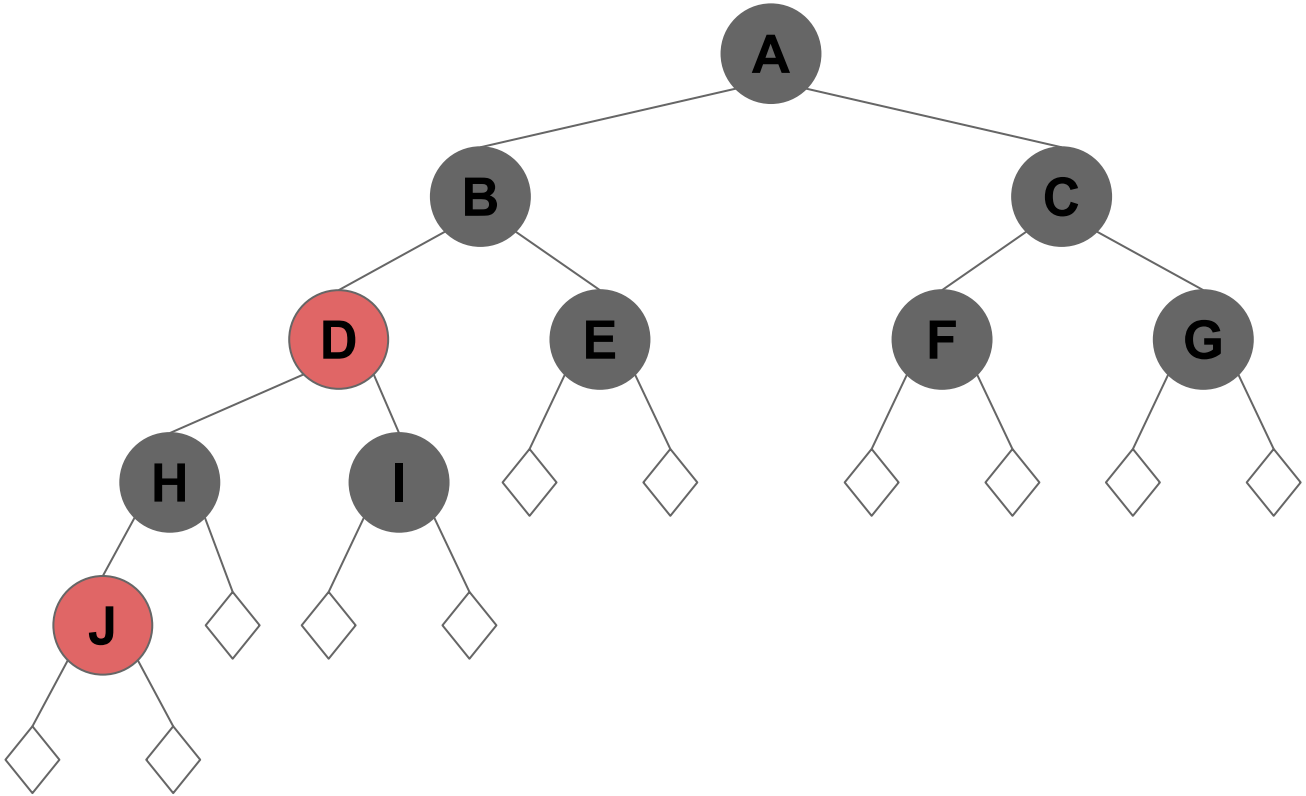
To Enforce the Depth Constraint on empty nodes:

1. Color each node red or black
   a. The # of black nodes from each empty node to root must be same
   b. The parent of a red node must always be black
2. On insertion (or deletion)
   a. Inserted node
   b. Repair violati
      i. Make su

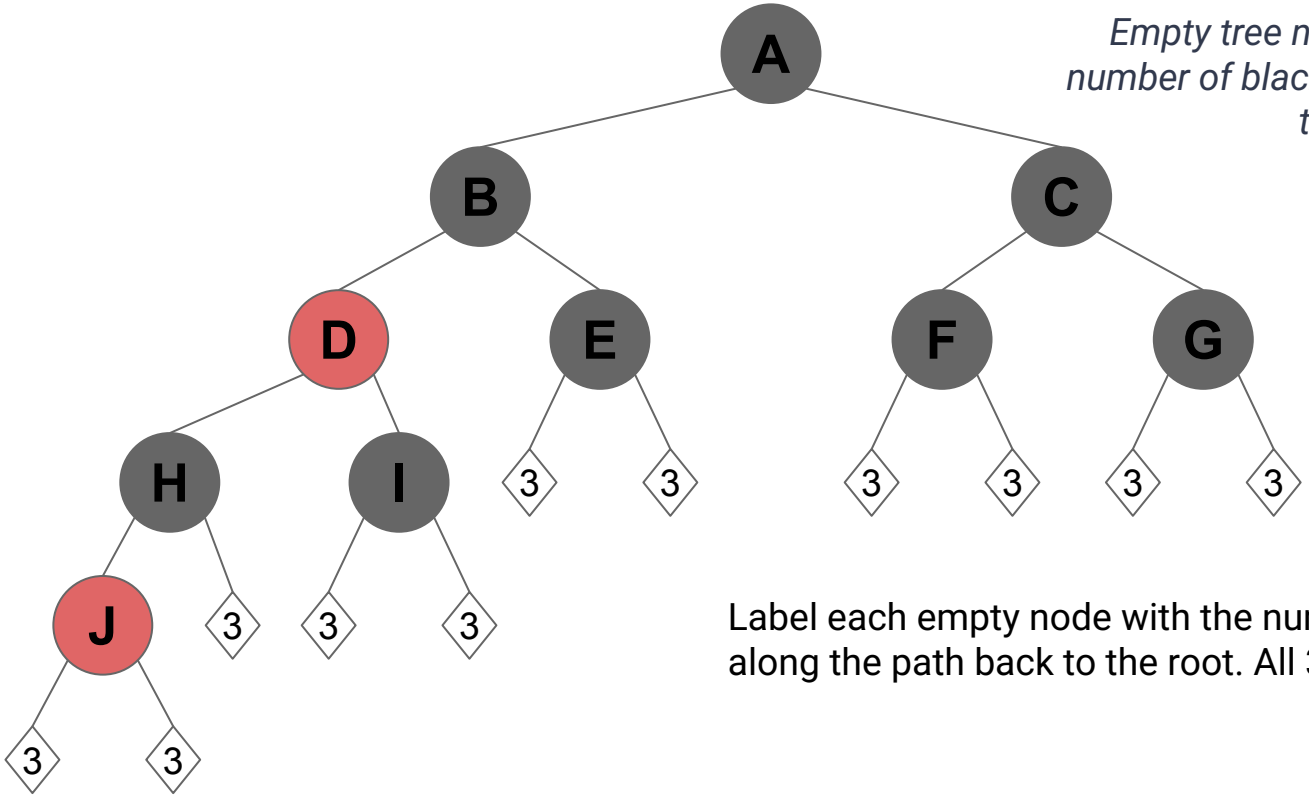
IMPORTANT: Just like with BSTs and AVL Trees, these constraints must hold true for EVERY node in the tree.

AKA every subtree in a Red-Black tree must also be a Red-Black Tree!

Red-Black Trees



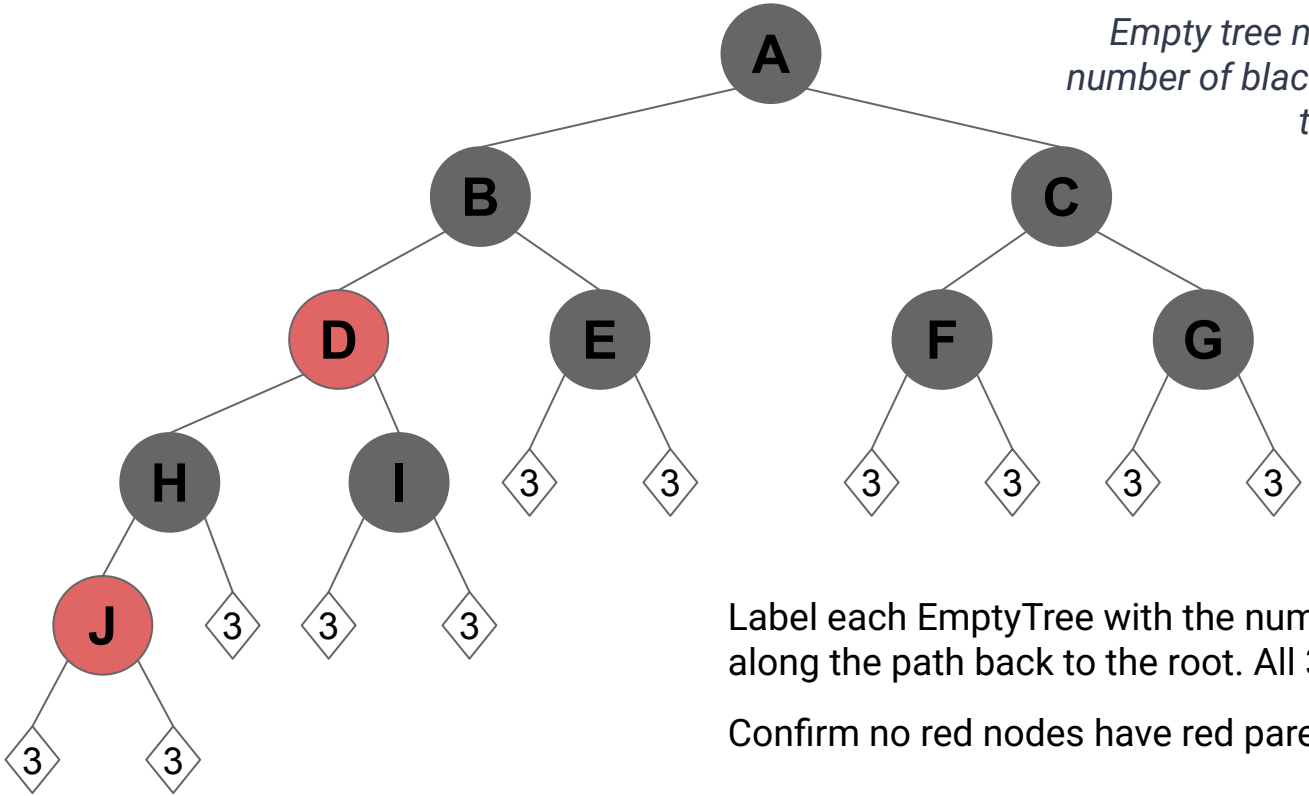
Red-Black Trees



Empty tree nodes are labeled with the number of black nodes on the path back to the root in this diagram

Label each empty node with the number of black nodes along the path back to the root. All 3 in this case ✓

Red-Black Trees



Empty tree nodes are labeled with the number of black nodes on the path back to the root in this diagram

Label each EmptyTree with the number of black nodes along the path back to the root. All 3 in this case ✓

Confirm no red nodes have red parents ✓

Red-Black Trees

How does this coloring relate to our depth constraint?

Red-Black Trees

Assume we have a valid Red-Black tree with X black nodes from on each path from empty node to root

What is the shallowest possible depth of an empty node?

Red-Black Trees

Assume we have a valid Red-Black tree with X black nodes from on each path from empty node to root

What is the shallowest possible depth of an empty node?

X black nodes in a row = X

Red-Black Trees

Assume we have a valid Red-Black tree with X black nodes from on each path from empty node to root

What is the shallowest possible depth of an empty node?

X black nodes in a row = X

What is the deepest possible depth of an empty node?

Red-Black Trees

Assume we have a valid Red-Black tree with X black nodes from on each path from empty node to root

What is the shallowest possible depth of an empty node?

X black nodes in a row = X

What is the deepest possible depth of an empty node?

X black nodes with 1 red node between each one = 2X

Red-Black Trees

Now we have:

1. If we color nodes red and black with the rules described, then the shallowest empty node will be at least half the depth of the deepest
2. If the shallowest empty node is at least half the depth of the deepest then the depth of our tree is O(log(n))

Red-Black Trees

Now we have:

1. If we color nodes red and black with the rules described, then the shallowest empty node will be at least half the depth of the deepest
2. If the shallowest empty node is at least half the depth of the deepest then the depth of our tree is O(log(n))

So how do we build/color our tree?

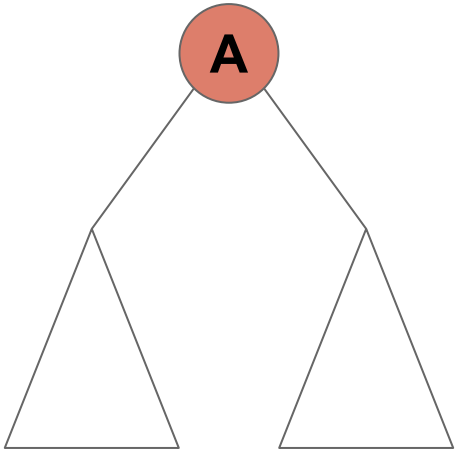
Red-Black Trees

After insertion or deletion, what situations can we encounter?

Red-Black Trees

After insertion or deletion, what situations can we encounter?

Case 1a: Our root is red, we're all good! ✓

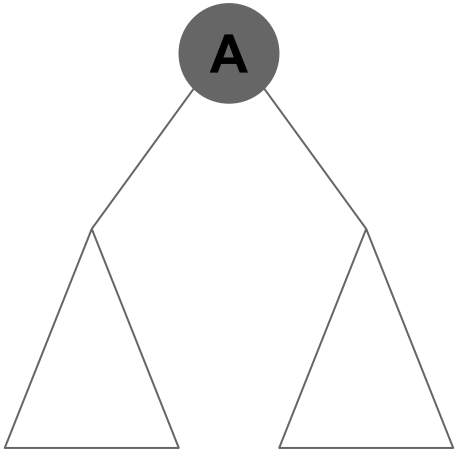


Triangles represent **valid** Red-Black tree fragments

Red-Black Trees

After insertion or deletion, what situations can we encounter?

Case 1b: Our root is black, we're all good! ✓

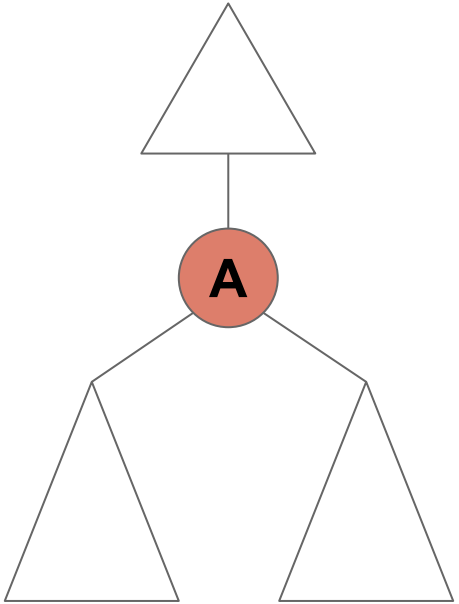


Triangles represent **valid** Red-Black tree fragments

Red-Black Trees

After insertion or deletion, what situations can we encounter?

Case 2: The node we are checking is red…

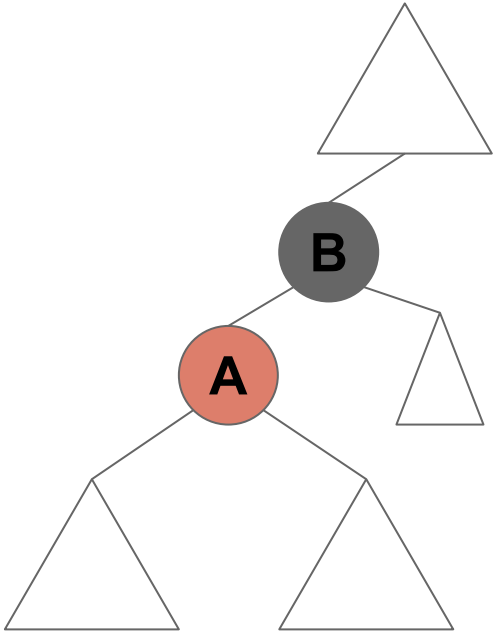


Triangles represent **valid** Red-Black tree fragments

Red-Black Trees

After insertion or deletion, what situations can we encounter?

Case 2: The node we are checking is red… and it's parent is black. We are all good! ✓

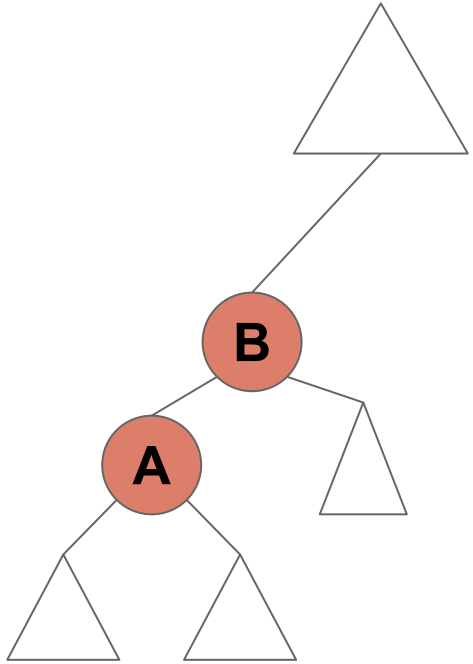


Triangles represent **valid** Red-Black tree fragments

Red-Black Trees

After insertion or deletion, what situations can we encounter?

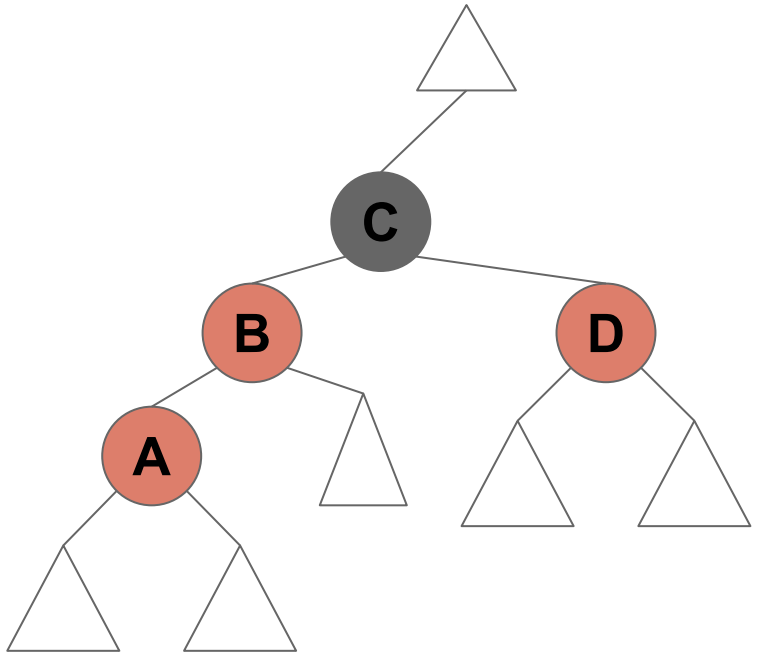
Case 3: The node we are checking is red… and it's parent is red. Now we have to fix the tree.



Red-Black Trees

After insertion or deletion, what situations can we encounter?

Case 3a: The node we are checking is red… and it's parent is red. That node's parent is black and it's sibling is red…

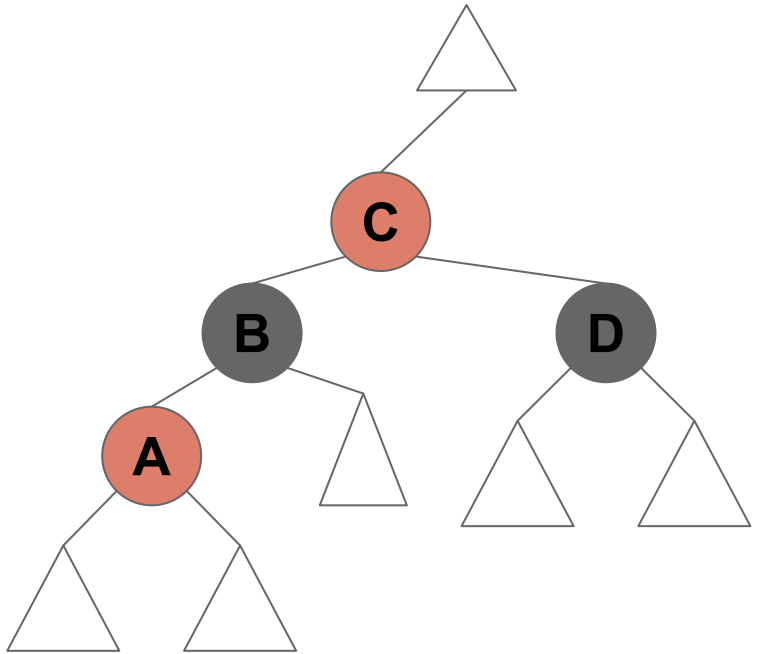


Red-Black Trees

After insertion or deletion, what situations can we encounter?

Case 3a: The node we are checking is red… and it's parent is red. That node's parent is black and it's sibling is red…

Recolor B,C,D. Are we all good?

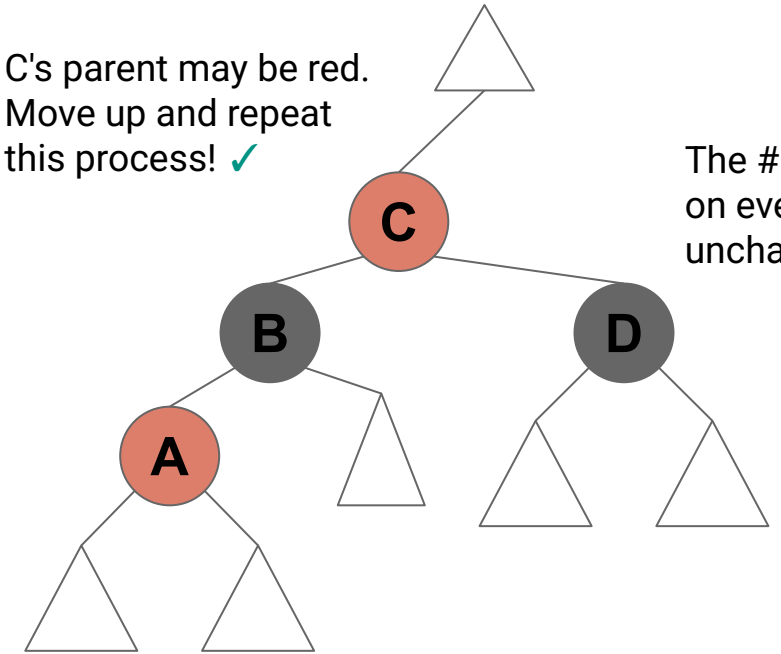


Red-Black Trees

After insertion or deletion, what situations can we encounter?

Case 3a: The node we are checking is red… and it's parent is red. That node's parent is black and it's sibling is red…

Recolor B,C,D. Are we all good?



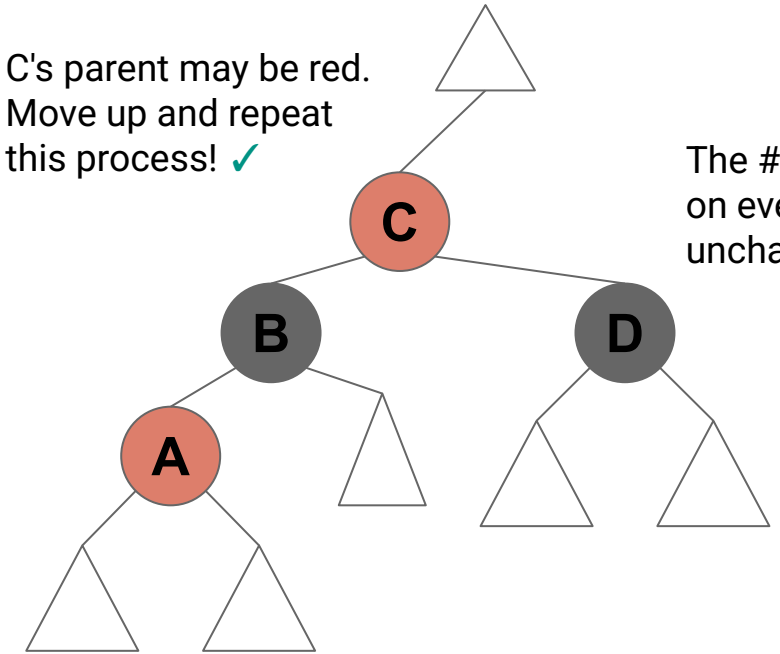
Red-Black Trees

After insertion or deletion, what situations can we encounter?

Case 3a: The node we are checking is red… and it's parent is red. That node's parent is black and it's sibling is red…

Recolor B,C,D. Are we all good?

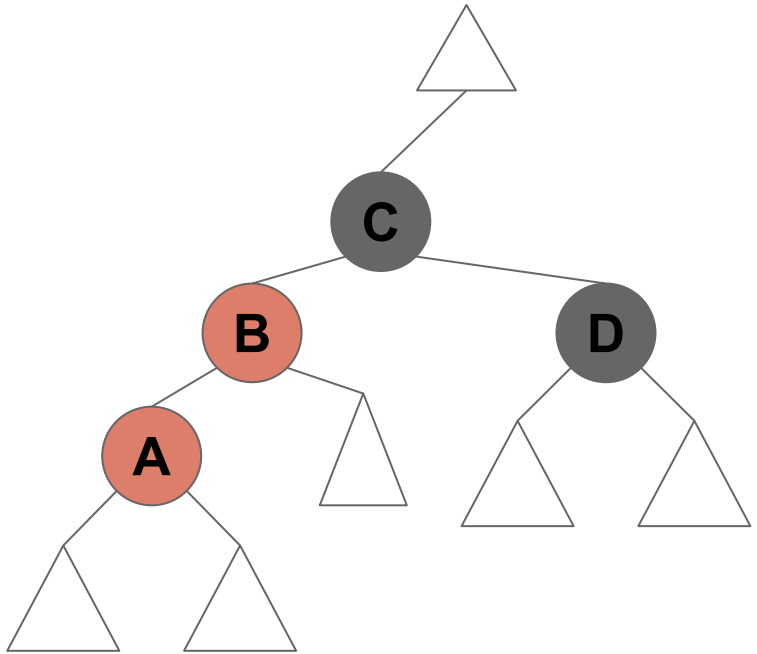
Note: This also works if A is right child of B and/or B is right child of C



Red-Black Trees

After insertion or deletion, what situations can we encounter?

Case 3b: The node we are checking is red… and it's parent is red. That node's parent is black and it's sibling is black…

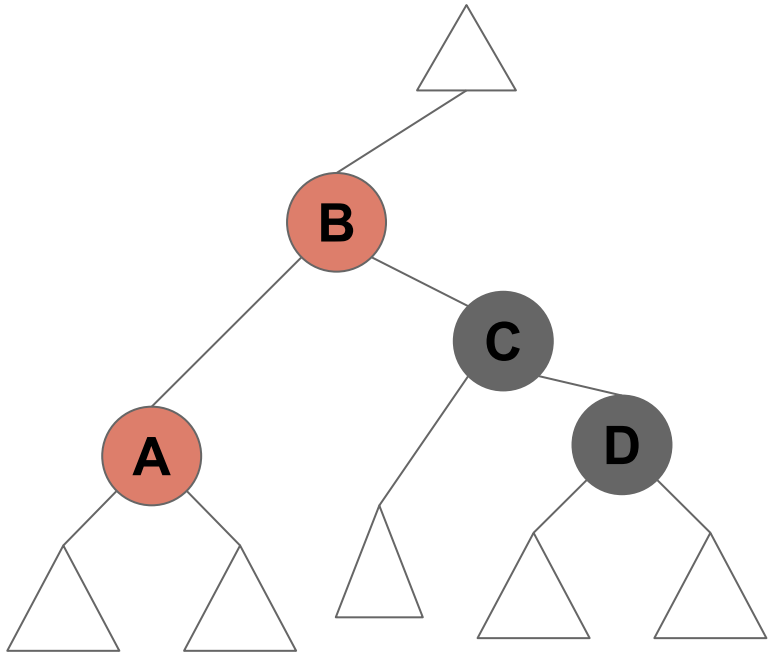


Red-Black Trees

After insertion or deletion, what situations can we encounter?

Case 3b: The node we are checking is red… and it's parent is red. That node's parent is black and it's sibling is black…

Rotate(B,C)



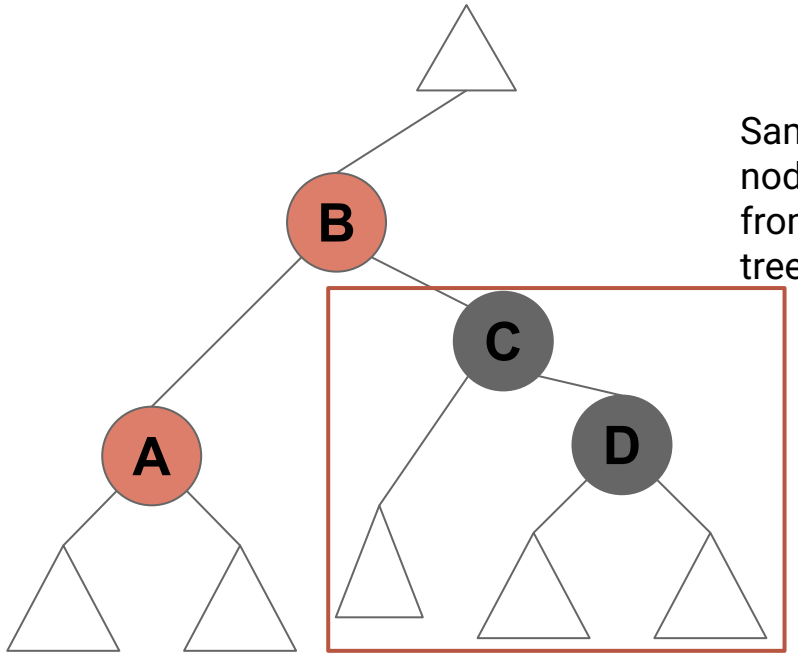
Red-Black Tree

After insertion or deletion, what situations can we encounter?

Case 3b: The node we are checking is red… and it's parent is red. That node's parent is black and it's sibling is black…

Rotate(B,C)

1 less black node to root for this part of the tree…



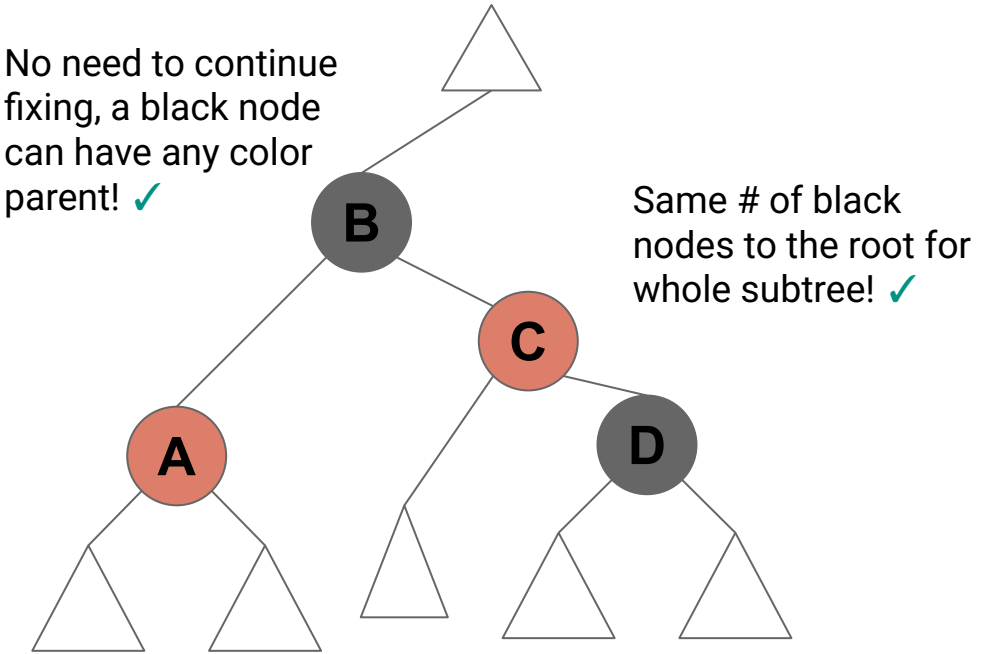
Same # of black nodes to the root from this part of tree

Red-Black Trees

After insertion or deletion, what situations can we encounter?

Case 3b: The node we are checking is red… and it's parent is red. That node's parent is black and it's sibling is black…

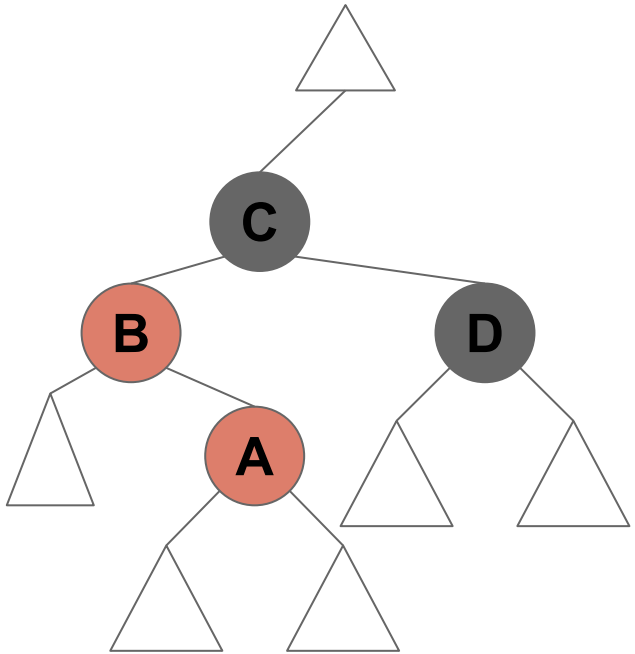
Rotate(B,C)
Recolor(B,C)



Red-Black Trees

After insertion or deletion, what situations can we encounter?

Case 3c: The node we are checking is red… and it's parent is red. That node's parent is black and it's sibling is black…but A is the right child of B

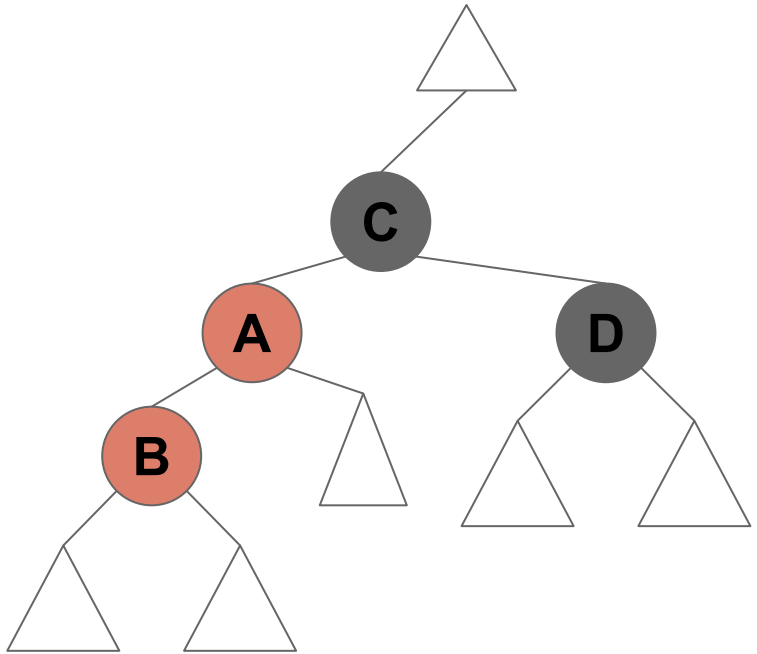


Red-Black Trees

After insertion or deletion, what situations can we encounter?

Case 3c: The node we are checking is red… and it's parent is red. That node's parent is black and it's sibling is black…but A is the right child of B

Rotate(B,A) now we are back to **3b**



Red-Black Trees

Note: Each insertion creates at most one red-red parent-child conflict

- O(1) time to recolor/rotate to repair the parent-child conflict
- May create a red-red conflict in grandparent
  - Up to d/2 = O(log(n)) repairs required, but each repair is O(1)
- **Insertion therefore remains O(log(n))**

Note: Each deletion removes at most one black node (red doesn't matter)

- O(1) time to recolor/rotate to preserve black-depth
- May require recoloring (grand-)parent from black to red
  - Up to d = O(log(n)) repairs required
- **Deletion therefore remains O(log(n))**

BST Operations

Operation	BST	AVL	Red-Black
`find`	$O(d) = O(n)$	$O(d) = O(\log n)$	$O(d) = O(\log n)$
`insert`	$O(d) = O(n)$	$O(d) = O(\log n)$	$O(d) = O(\log n)$
`remove`	$O(d) = O(n)$	$O(d) = O(\log n)$	$O(d) = O(\log n)$

The tree operations on a BST are always $O(d)$ (they involve a constant number of trips from root to leaf at most).

The balanced varieties (AVL and Red-Black) constrain the depth