CSE 250
Data Structures

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

Lec 28: Red-Black Trees

Announcements

- PA2 due yesterday, submissions close tomorrow
- WA4 released today
  - It is due after the midterm, but starting it early will help you study!

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 Trees

An **AVL tree** (**A**delson-**V**elsky and **L**andis) is a ***BST*** where every subtree is depth-balanced

Remember: Tree depth = height(root)

Balanced: |height(root.right) - height(root.left)| ≤ 1

AVL Trees

Define **`balance(`*`v`*`) = height(`*`v.right`*`) - height(`*`v.left`*`)`**

Goal: Maintaining **`balance(`*`v`*`)`** $\in$ **{ -1, 0, 1 }**

- **`balance(`*`v`*`) = 0`** → "***v*** is balanced"
- **`balance(`*`v`*`) = -1`** → "***v*** is left-heavy"
- **`balance(`*`v`*`) = 1`** → "***v*** is right-heavy"

An Important Note About Height!

The height of a tree is the number of edges that need to be followed to get to the deepest leaf

- Therefore the depth of a single node tree is 0
- As a convention, the depth of an empty tree is -1

AVL Trees - Depth Bounds

Question: Does the AVL property result in any guarantees about depth?

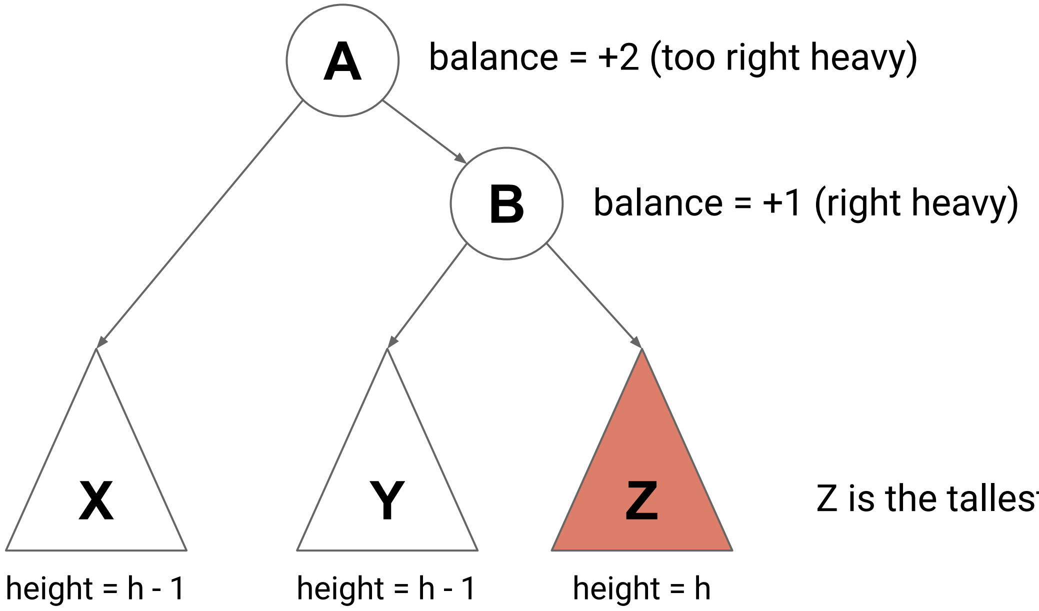
YES! Depth balance forces a maximum possible depth of **log(*n*)**

AVL Trees - Enforcing the Depth Bound

Key Observations:

- Adding a node to an AVL tree can increase subtree height by at most 1
- Removing a node can decrease subtree height by at most 1
- Both of these modifications only affect ancestors
- A rotation maintains ordering, and changes tree height by at most +/-1

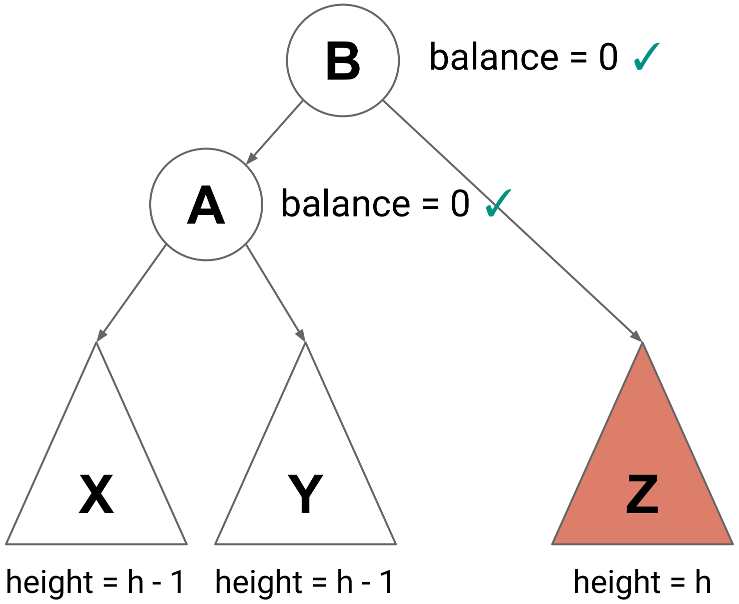
Enforcing the AVL Constraint: Case 1



How can we fix this?

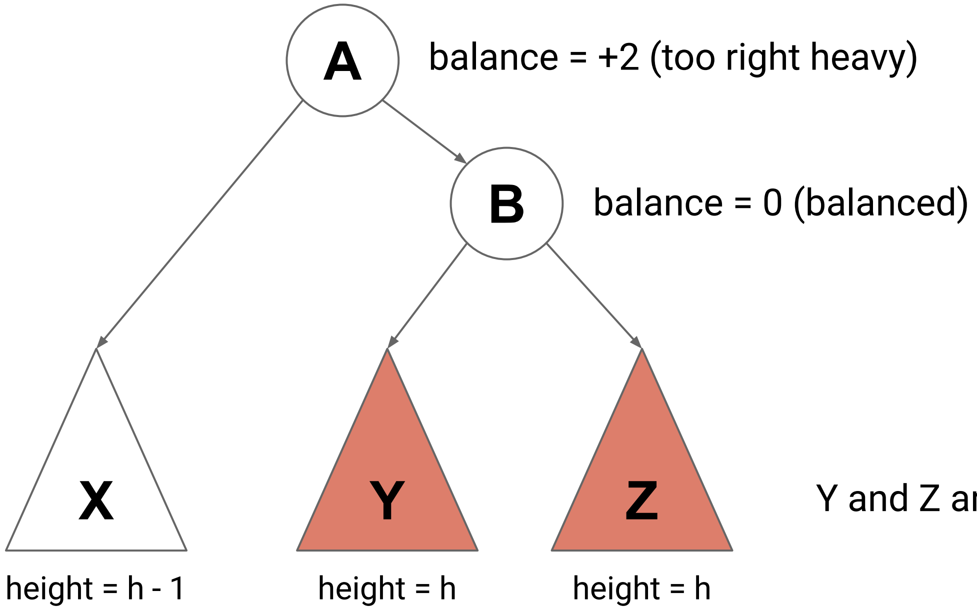
Z is the tallest of X, Y, Z

Enforcing the AVL Constraint: Case 1



How can we fix this? **`rotate(A,B)`**

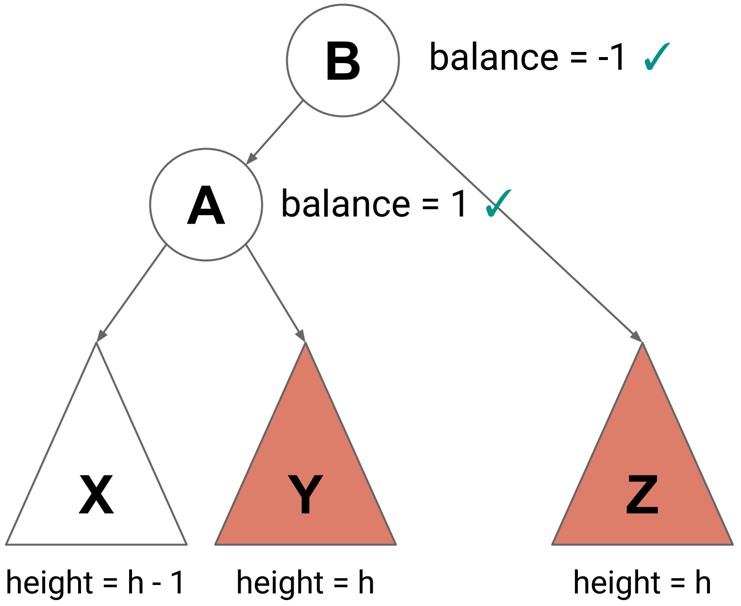
Enforcing the AVL Constraint: Case 2



How can we fix this?

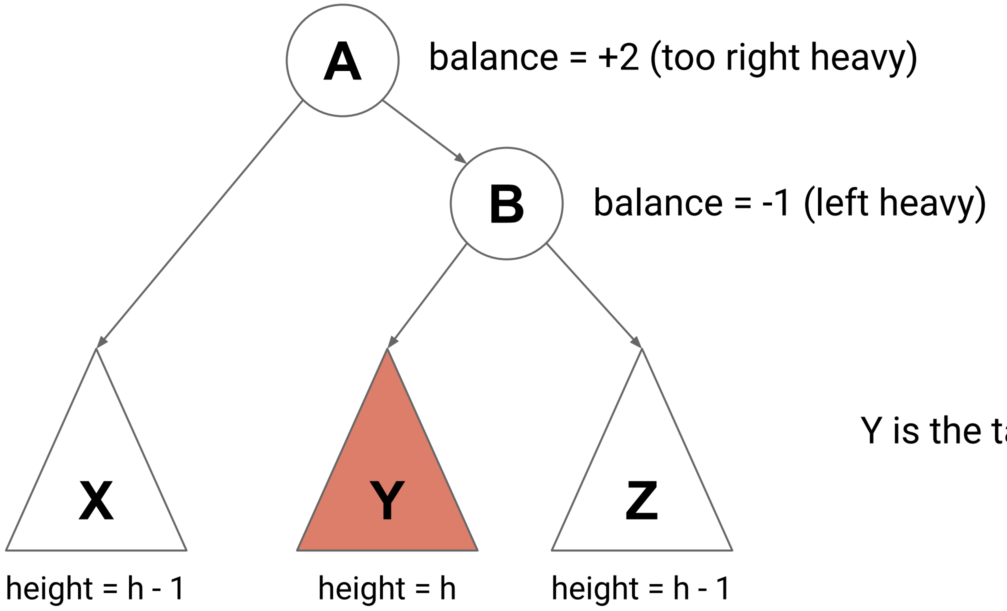
Y and Z are taller than X

Enforcing the AVL Constraint: Case 2



How can we fix this? **`rotate(A,B)`**

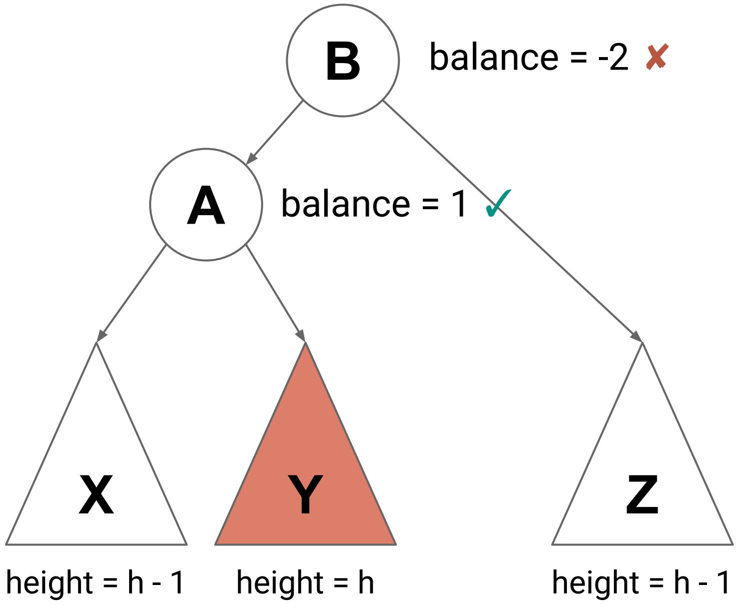
Enforcing the AVL Constraint: Case 3



How can we fix this?

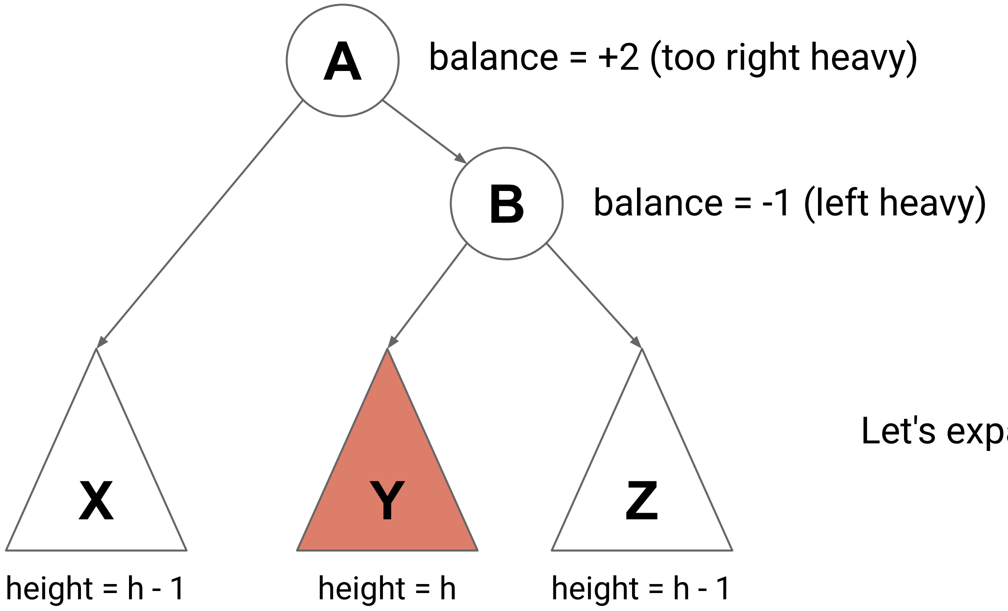
Y is the tallest of X, Y, Z

Enforcing the AVL Constraint: Case 3



How can we fix this?
Will just a single left rotation work? **No**

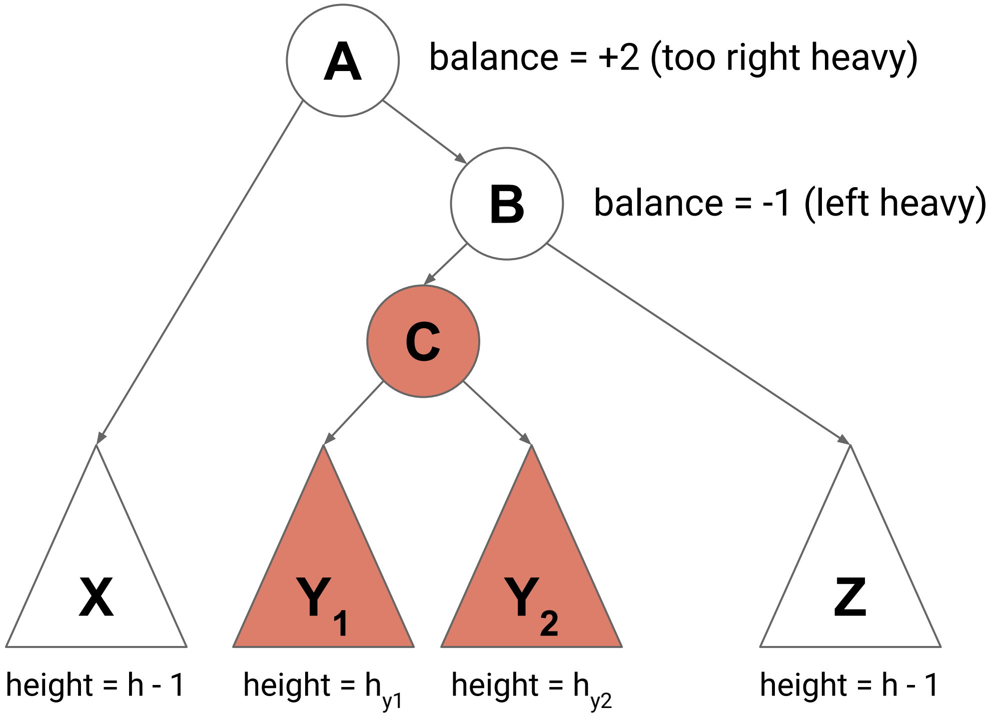
Enforcing the AVL Constraint: Case 3



How can we fix this?
Will just a single left rotation work? **No**

Let's expand Y to figure out what to do

Enforcing the AVL Constraint: Case 3



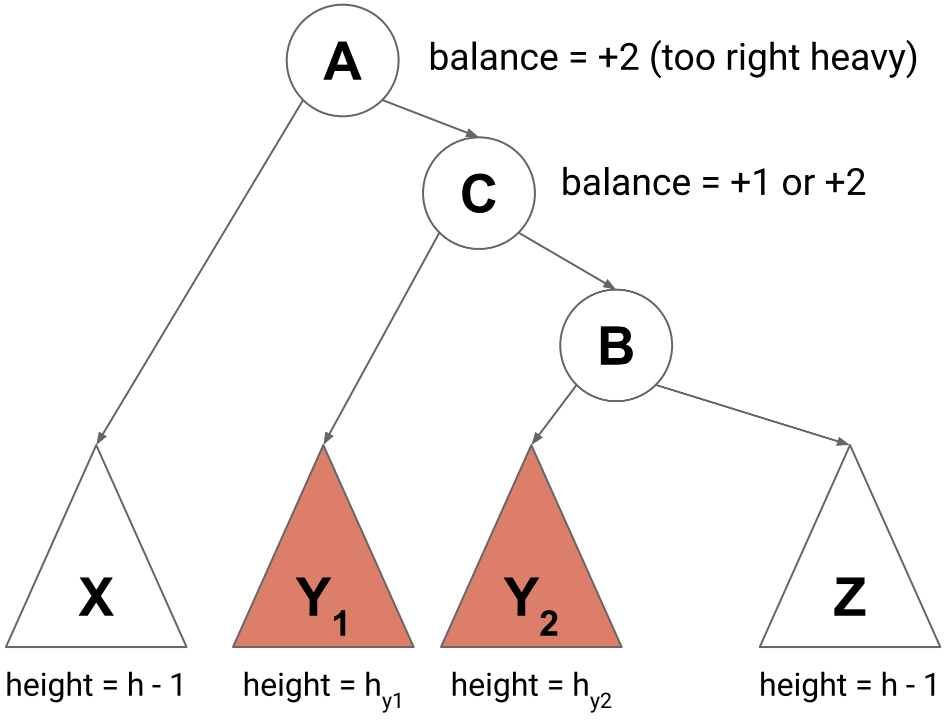
How can we fix this?

Height of **C** we know must be ***h***

Therefore At least one of $\boldsymbol{h_{y1}}$ or $\boldsymbol{h_{y2}}$ must be ***h* - 1**

The other can be ***h* - 2**, or ***h* - 1**

Enforcing the AVL Constraint: Case 3



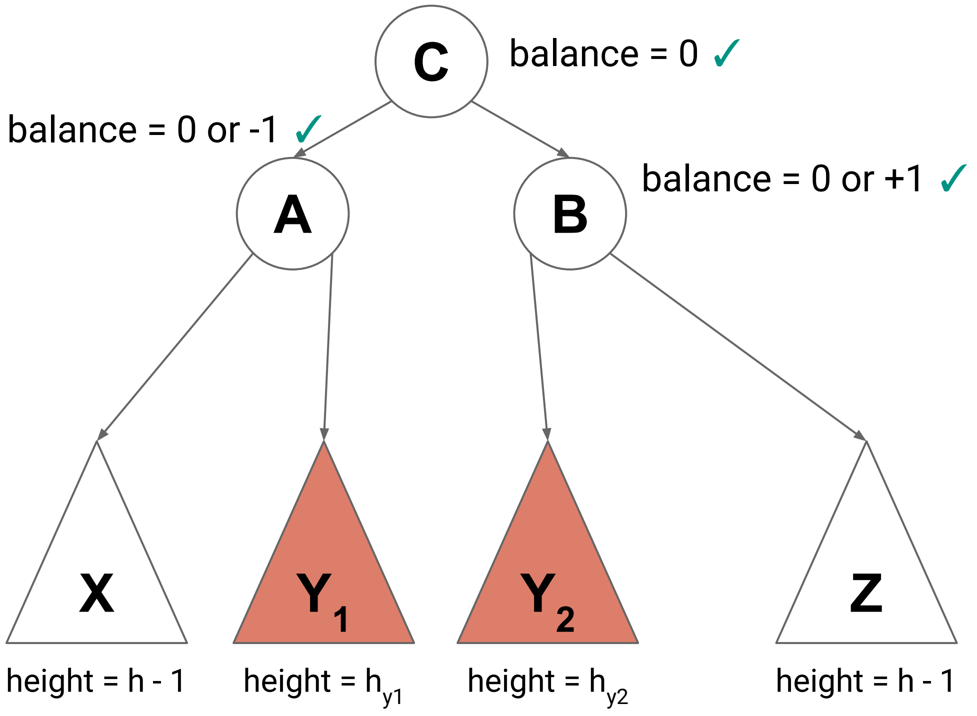
How can we fix this?
Rotate right first: **`rotate(B,C)`**

Height of **C** we know must be ***h***

Therefore At least one of $\boldsymbol{h_{y1}}$ or $\boldsymbol{h_{y2}}$ must be ***h* - 1**

The other can be ***h* - 2**, or ***h* - 1**

Enforcing the AVL Constraint: Case 3



How can we fix this?
Rotate right first: **`rotate(B,C)`**
Then right left: **`rotate(A,C)`**

Height of **C** we know must be ***h***

Therefore At least one of $\boldsymbol{h_{y1}}$ or $\boldsymbol{h_{y2}}$ must be ***h* - 1**

The other can be ***h* - 2**, or ***h* - 1**

Enforcing the AVL Constraint

- If too right heavy (balance == +2)
    - If right child is right heavy (balance == +1) or balanced (balance == 0)
        - rotate left around the root
    - If right child is left heavy (balance == -1)
        - rotate right around right child, then rotate left around root
- If too left heavy (balance == -2)
    - Same as above but flipped

Therefore if we have a balance factor that is off, but all children are AVL trees, we can fix the balance factor in at most 2 rotations

Inserting Records

To insert a record into an AVL Tree:

1. Find the insertion point (remember it is a BST)	$\boldsymbol{O(d) = O(\log n)}$
2. Insert the new leaf and set balance factor to 0	$\boldsymbol{O(1)}$
3. Trace path back up to root and update balance factors	$\boldsymbol{O(d) = O(\log n)}$
a. If a balance factor becomes +/-2 then rotate to fix	$\boldsymbol{O(1)}$

Inserting New Nodes

```java
public void insert(T value, AVLTreeNode<T> root) {
  // Use normal logic for inserting into a BST, then set heavy flags
  AVLTreeNode<T> newNode = insertIntoBST(value, root);
  newNode.isLeftHeavy = newNode.isRightHeavy = false;
  while (newNode.parent.isPresent()) {
    if (newNode.parent.get().leftChild.orElse(null) == newNode) {
      // Fix issues that occur from inserting into parents left subtree
    } else {
      // Fix issues that occur from inserting into parents right subtree
    }
    newNode = newNode.parent.get();
  }
}
```

Inserting New Nodes

```java
public void insert(T value, AVLTreeNode<T> root) {
  // Use normal logic for inserting into a BST, then set heavy flags
  AVLTreeNode<T> newNode = insertIntoBST(value, root);
  newNode.isLeftHeavy = newNode.isRightHeavy = false;
  while (newNode.parent.isPresent()) {
    if (newNode.parent.get().leftChild.orElse(null) == newNode) {
      // Fix issues that occur from inserting into parents left subtree
    } else {
      // Fix issues that occur from inserting into parents right subtree
    }
    newNode = newNode.parent.get();
  }
}
```

Find insertion point and create the new leaf $O(d) = O(\log n)$

Inserting New Nodes

```java
public void insert(T value, AVLTreeNode<T> root) {
  // Use normal logic for inserting into a BST, then set heavy flags
  AVLTreeNode<T> newNode = insertIntoBST(value, root);
  newNode.isLeftHeavy = newNode.isRightHeavy = false;
  while (newNode.parent.isPresent()) {
    if (newNode.parent.get().leftChild.orElse(null) == newNode) {
      // Fix issues that occur from inserting into parents left subtree
    } else {
      // Fix issues that occur from inserting into parents right subtree
    }
    newNode = newNode.parent.get();
  }
}
```

$O(d) = O(\log n)$ iterations

Inserting New Nodes

```java
public void insert(T value, AVLTreeNode<T> root) {
  // Use normal logic for inserting into a BST, then set heavy flags
  AVLTreeNode<T> newNode = insertIntoBST(value, root);
  newNode.isLeftHeavy = newNode.isRightHeavy = false;
  while (newNode.parent.isPresent()) {
    if (newNode.parent.get().leftChild.orElse(null) == newNode) {
      // Fix issues that occur from inserting into parents left subtree
    } else {
      // Fix issues that occur from inserting into parents right subtree
    }
    newNode = newNode.parent.get();
  }
}
```

What is the cost of each iteration?
How exactly do we fix the issues? (next slide)

Inserting New Nodes

```
if (newNode.parent.get().leftChild.orElse(null) == newNode) {
  // Fix issues that occur from inserting into parents left subtree
  if (newNode.parent.get().isRightHeavy) {
    newNode.parent.get().isRightHeavy = false;
    return
  } else if (newNode.parent.get().isLeftHeavy) {
    if (newNode.isLeftHeavy) newNode.parent.get().rotateRight();
    else newNode.parent.get().rotateLeftRight();
    return
  } else {
    newNode.parent.get().isLeftHeavy = true;
  }
}
```

Inserting New Nodes

```
if (newNode.parent.get().leftChild.orElse(null) == newNode) {
  // Fix issues that occur from inserting into parents left subtree
  if (newNode.parent.get().isRightHeavy) {
    newNode.parent.get().isRightHeavy = false;
    return
  } else if (newNode.parent.get().isLeftHeavy) {
    if (newNode.isLeftHeavy) newNode.parent.get().rotateRight();
    else newNode.parent.get().rotateLeftRight();
    return
  } else {
    newNode.parent.get().isLeftHeavy = true;
  }
}
```

If we inserted into the left of a right heavy subtree, then the subtree is no longer right heavy and we can stop here

Inserting New Nodes

```
if (newNode.parent.get().leftChild.orElse(null)
  // Fix issues that occur from inserting into
  if (newNode.parent.get().isRightHeavy) {
    newNode.parent.get().isRightHeavy = false;
    return
  } else if (newNode.parent.get().isLeftHeavy) {
    if (newNode.isLeftHeavy) newNode.parent.get().rotateRight();
    else newNode.parent.get().rotateLeftRight();
    return
  } else {
    newNode.parent.get().isLeftHeavy = true;
  }
}
```

If we inserted into the left of a left heavy subtree, then we just created imbalance, and need to rotate. But then we can stop.

Inserting New Nodes

```
if (newNode.parent.get().leftChild.orElse(null) == newNode) {
  // Fix issues that occur from inserting into parents left subtree
  if (newNode.parent.get().isRightHeavy) {
    newNode.parent.get().isRightHeavy = false;
    return
  } else if (newNode.parent.get().isLeftHeavy)
    if (newNode.isLeftHeavy) newNode.parent.get
    else newNode.parent.get().rotateLeftRight()
    return
  } else {
    newNode.parent.get().isLeftHeavy = true;
  }
}
```

If we inserted into the left of a balanced subtree, then we mark it as now being left heavy, and continue up the tree

Inserting New Nodes

```java
public void insert(T value, AVLTreeNode<T> root) {
  // Use normal logic for inserting into a BST, then set heavy flags
  AVLTreeNode<T> newNode = insertIntoBST(value, root);
  newNode.isLeftHeavy = newNode.isRightHeavy = false;
  while (newNode.parent.isPresent()) {
    if (newNode.parent.get().leftChild.orElse(null) == newNode) {
      // Fix issues that occur from inserting into parents left subtree
    } else {
      // Fix issues that occur from inserting into parents right subtree
    }
    newNode = newNode.parent.get();
  }
}
```

What is the cost of each iteration? **O(1)**

Inserting New Nodes

```java
public void insert(T value, AVLTreeNode<T> root) {
  // Use normal logic for inserting into a BST, then set heavy flags
  AVLTreeNode<T> newNode = insertIntoBST(value, root);
  newNode.isLeftHeavy = newNode.isRightHeavy = false;
  while (newNode.parent.isPresent()) {
    if (newNode.parent.get().leftChild.orElse(null) == newNode) {
      // Fix issues that occur from inserting into parents left subtree
    } else {
      // Fix issues that occur from inserting into parents right subtree
    }
    newNode = newNode.parent.get();
  }
}
```

Therefore, our total insertion cost is **$O(d) = O(\log(n))$**

Removing Records

- Removal follows essentially the same process as insertion
  - Do a normal BST removal
  - Go back up the tree adjusting balance factors
  - If you discover a balance factor that goes to +2/-2, rotate to fix

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 $\boldsymbol{d = O(\log(n))}$
- 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:

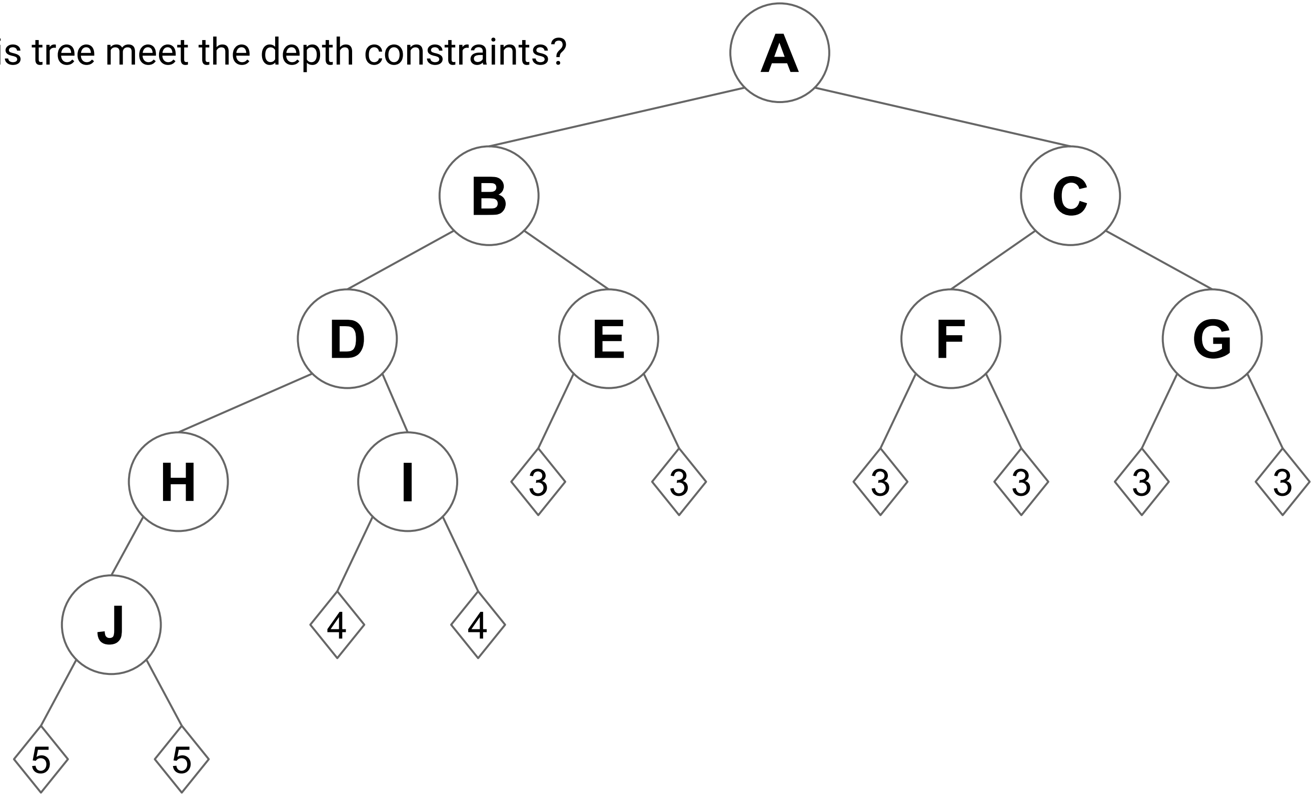
- Balance the depth of empty tree nodes
- If ***a***, ***b*** are EmptyTree nodes, then enforce that for all ***a***, ***b***:
  - depth(***a***) ≥ (depth(***b***) ÷ 2)

    or

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

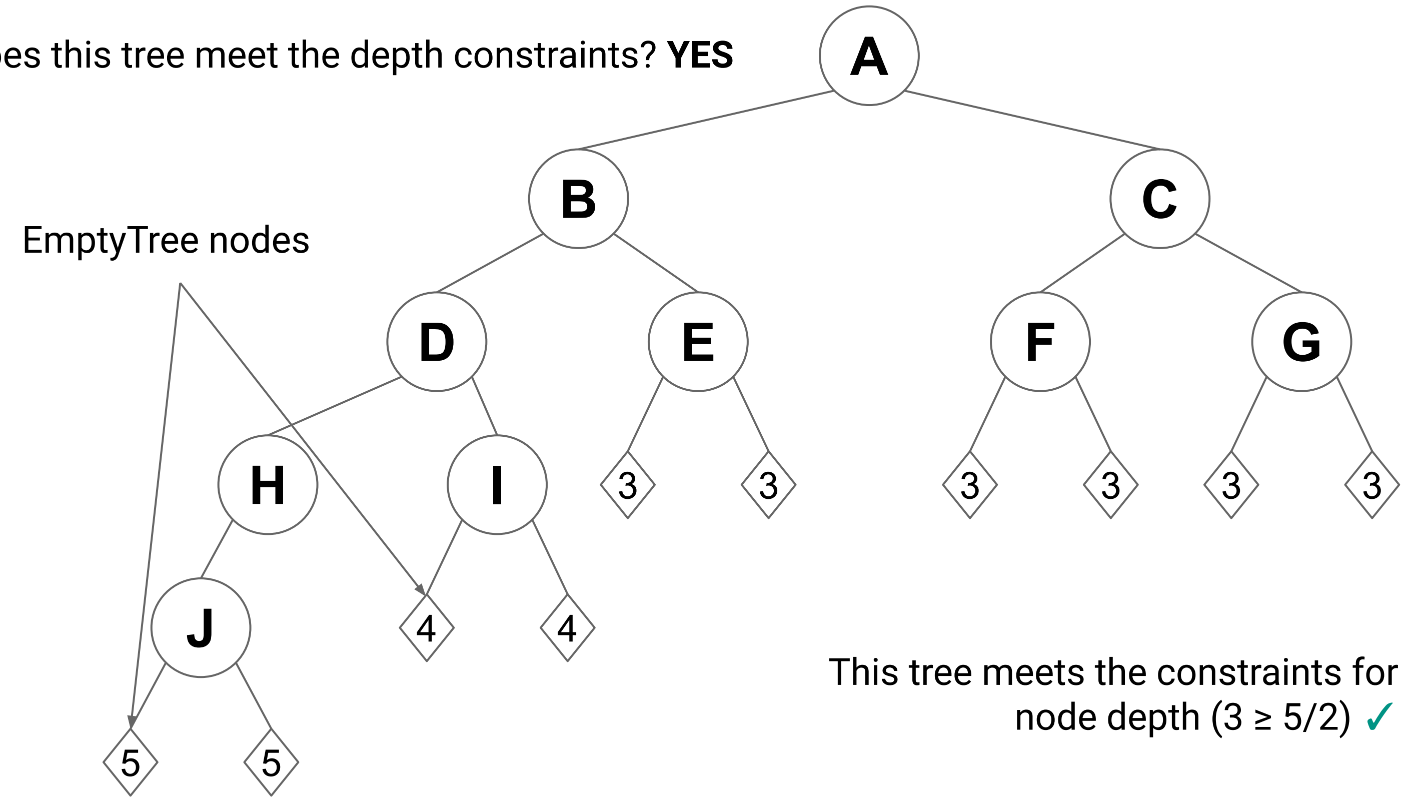
Depth Balancing

Does this tree meet the depth constraints?



Depth Balancing

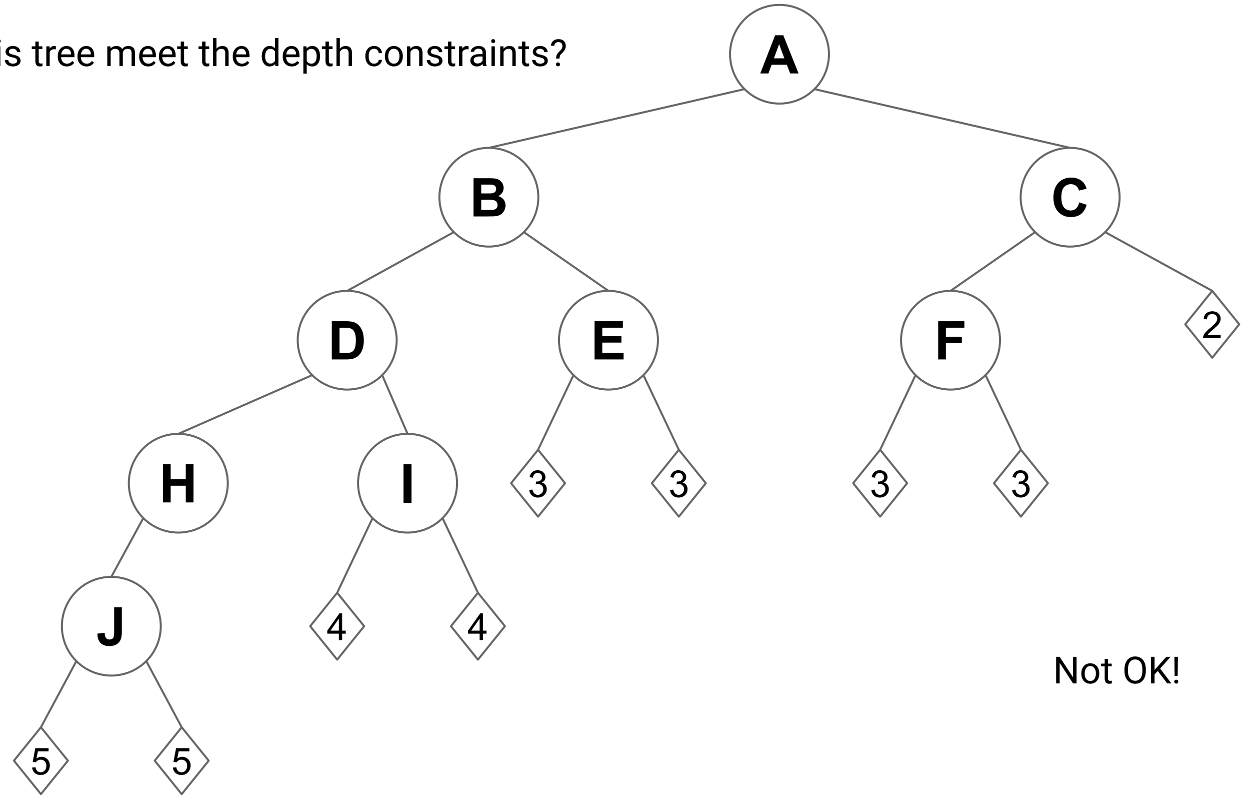
Does this tree meet the depth constraints? **YES**



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

Depth Balancing

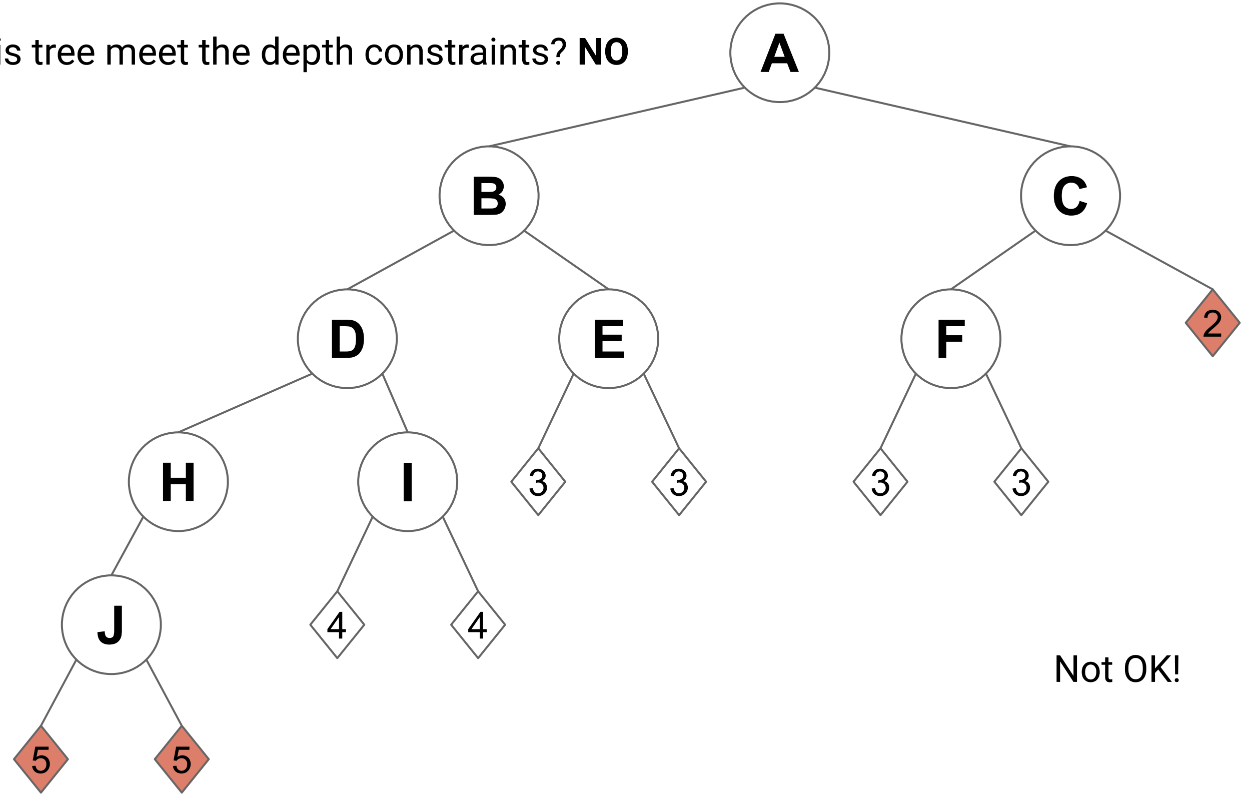
Does this tree meet the depth constraints?



Not OK!

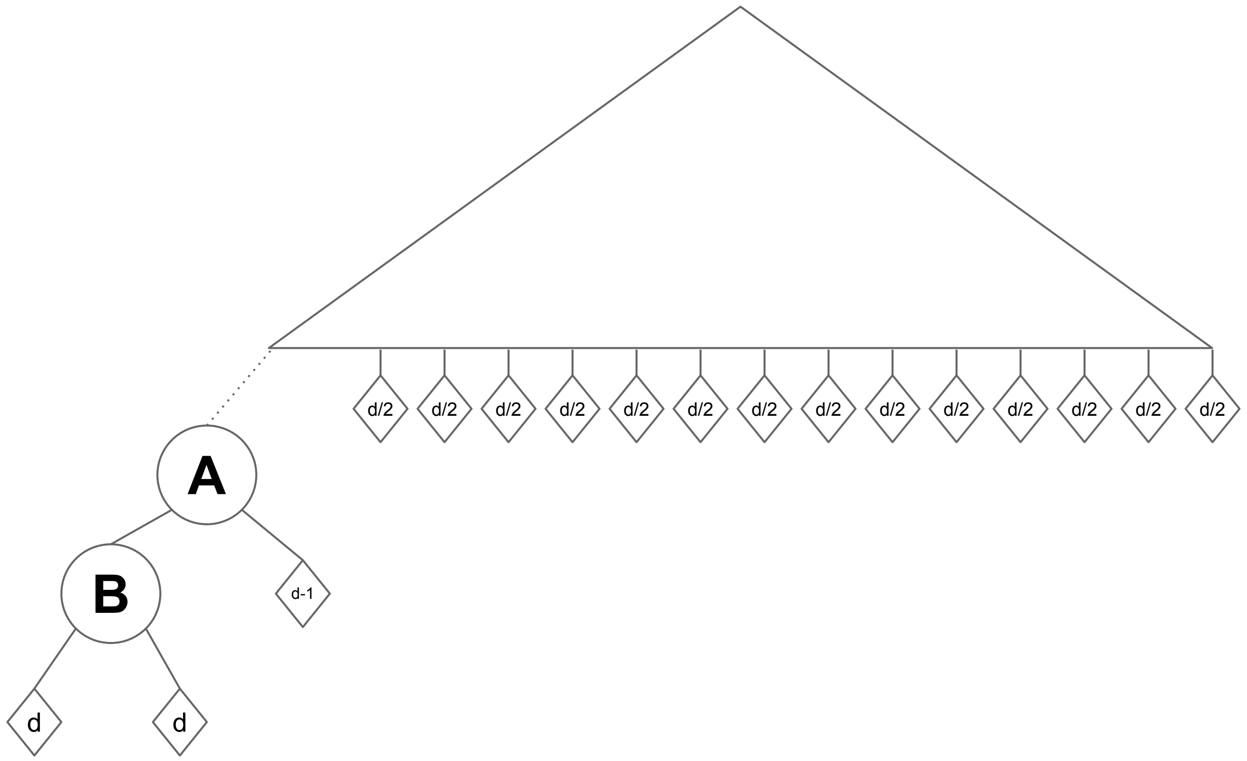
Depth Balancing

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

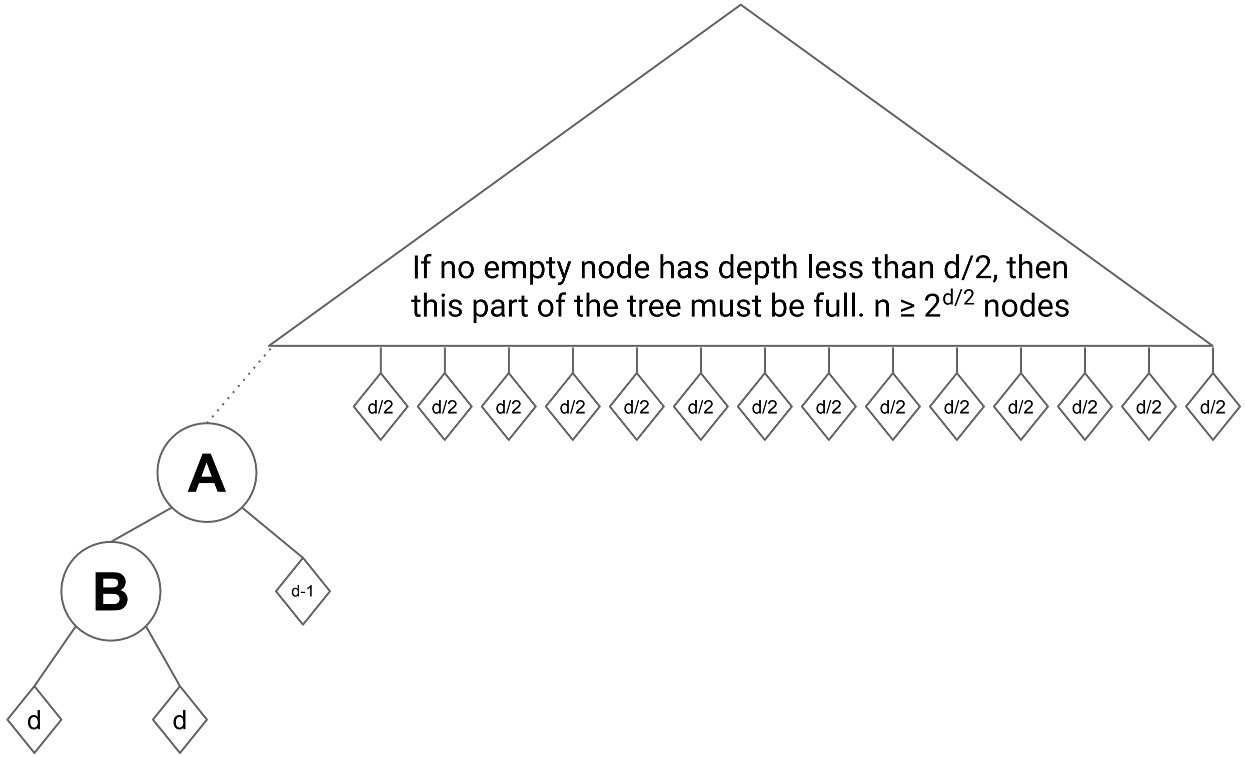


Not OK!

Depth Balancing

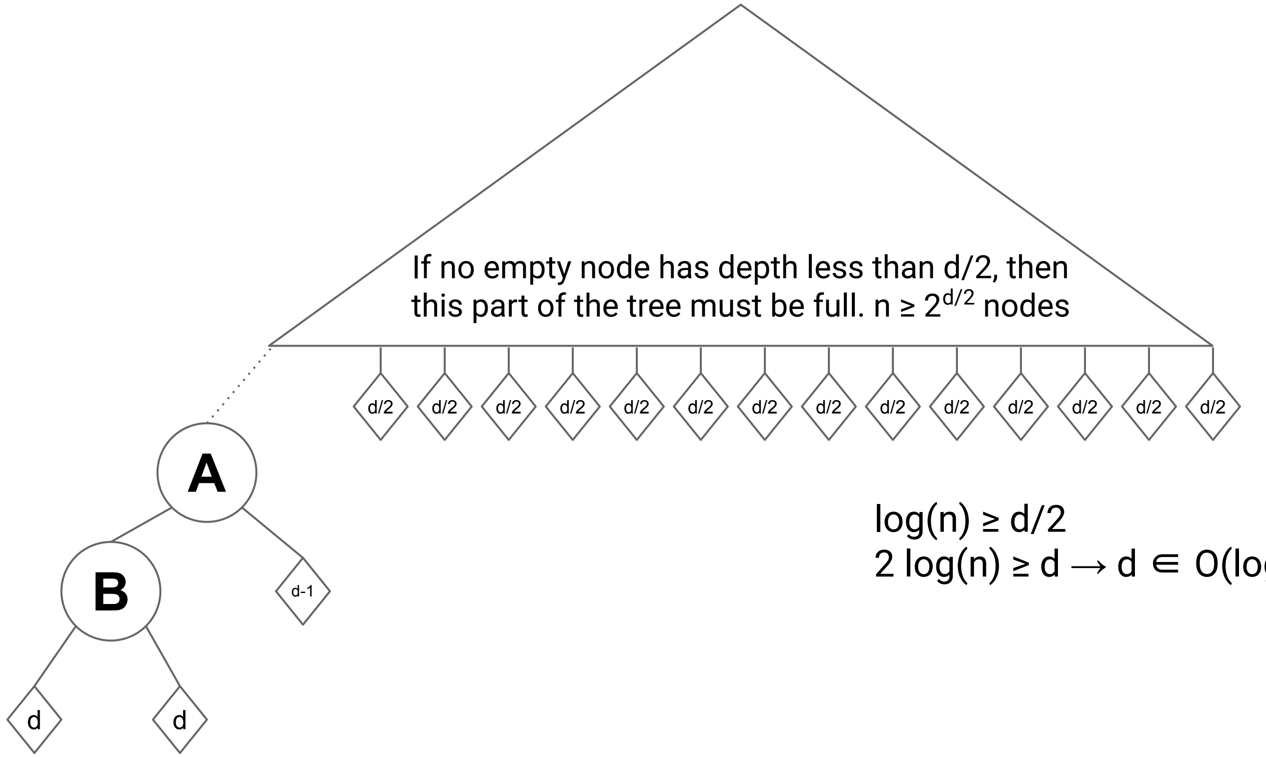


Depth Balancing



If no empty node has depth less than d/2, then this part of the tree must be full. n ≥ $2^{d/2}$ nodes
d/2
A
B
d-1
d
d

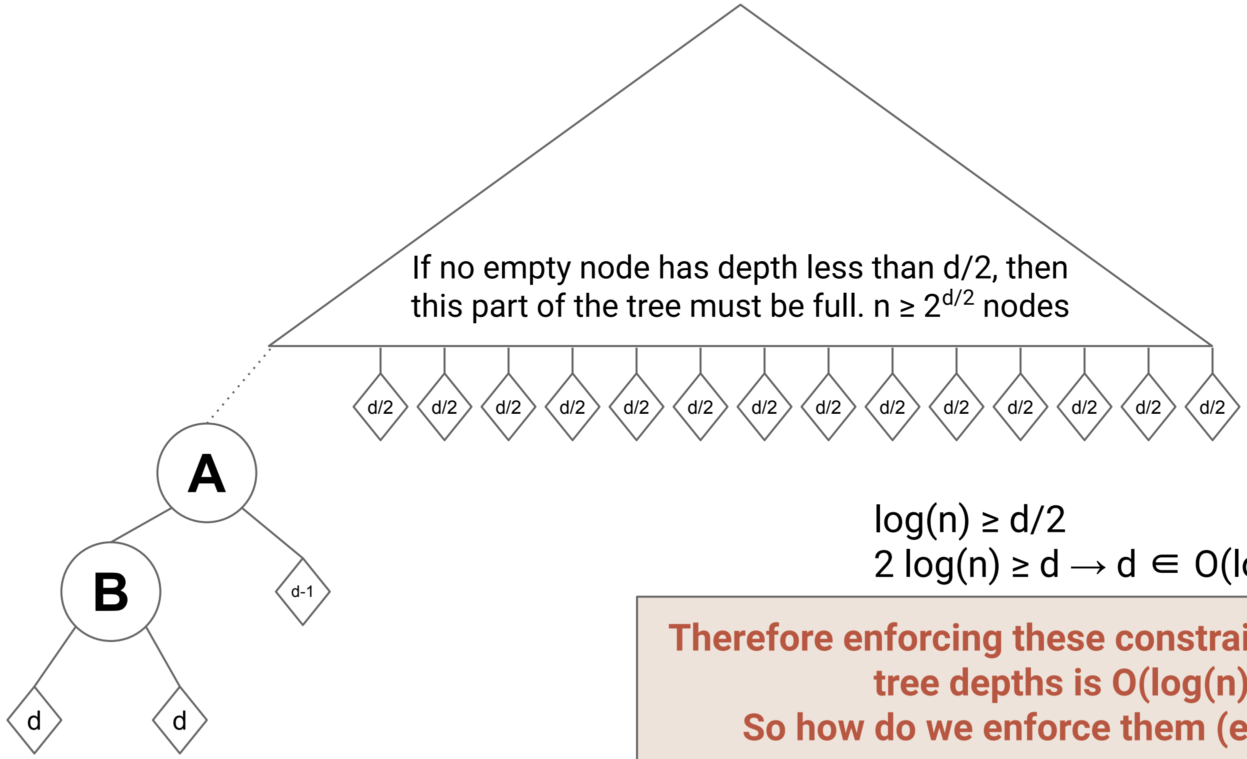
Depth Balancing


If no empty node has depth less than d/2, then this part of the tree must be full. n ≥ 2^{d/2} nodes
d/2
A
B
d-1
d
d

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

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

Depth Balancing


If no empty node has depth less than d/2, then
this part of the tree must be full. n ≥ 2^{d/2} nodes
d/2
d/2
d/2
d/2
d/2
d/2
d/2
d/2
d/2
d/2
d/2
d/2
d/2
d/2
A
B
d-1
d
d

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

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

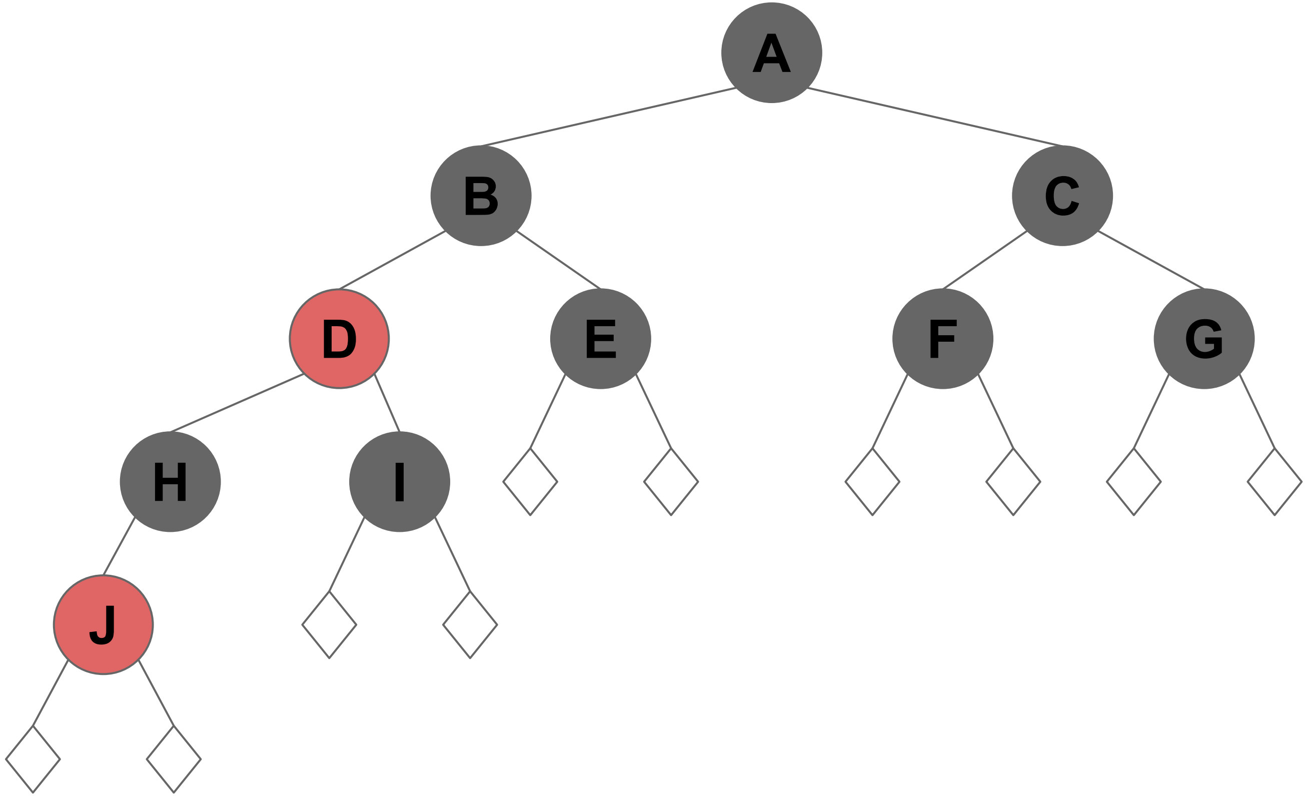
To Enforce the Depth Constraint on empty nodes:

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

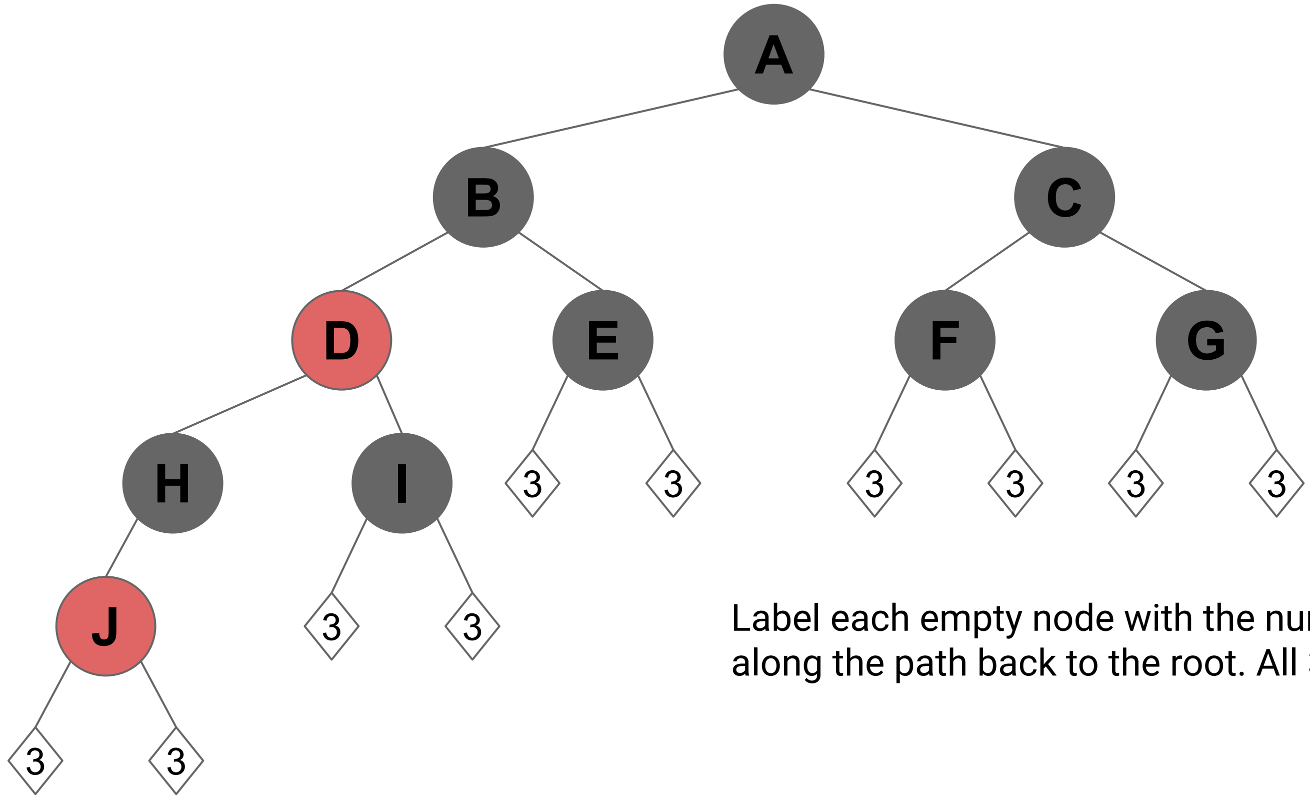
IMPORTANT: Just like with BSTs and AVL Trees, these constraints must hold true for EVERY node in the tree.

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

Red-Black Trees

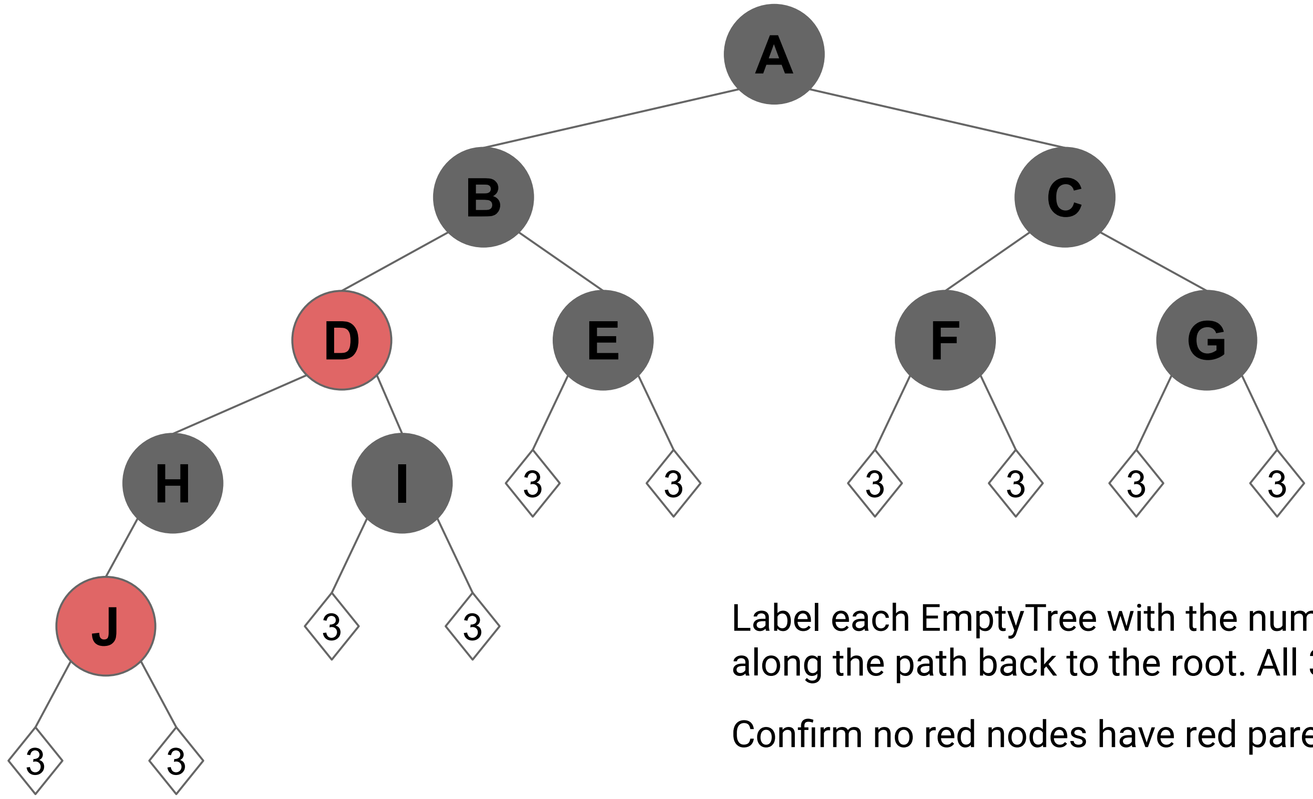


Red-Black Trees



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



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?

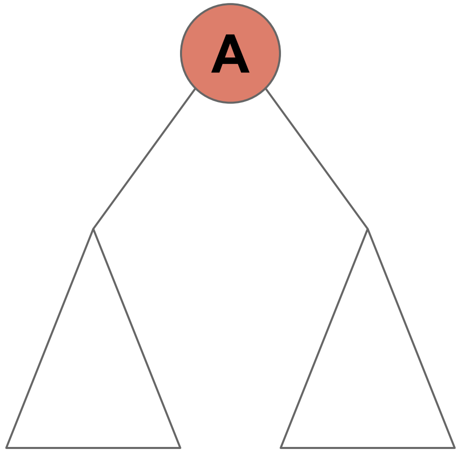
Red-Black Tree

After insertion or deletion, what situations can we encounter?

Red-Black Tree

After insertion or deletion, what situations can we encounter?

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

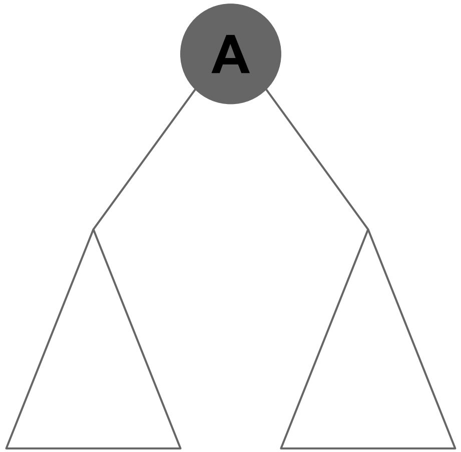


Triangles represent **valid** Red-Black tree fragments

Red-Black Tree

After insertion or deletion, what situations can we encounter?

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

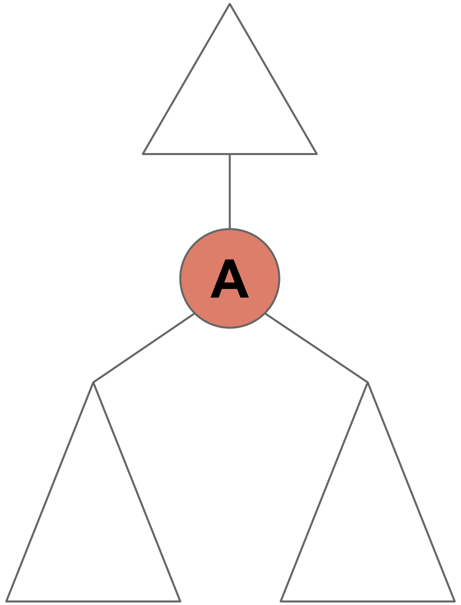


Triangles represent **valid** Red-Black tree fragments

Red-Black Tree

After insertion or deletion, what situations can we encounter?

Case 2: The node we are checking is red…

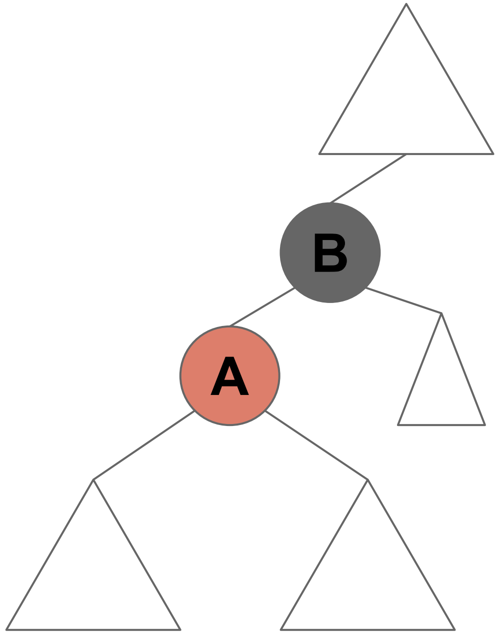


Triangles represent **valid** Red-Black tree fragments

Red-Black Tree

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! ✓

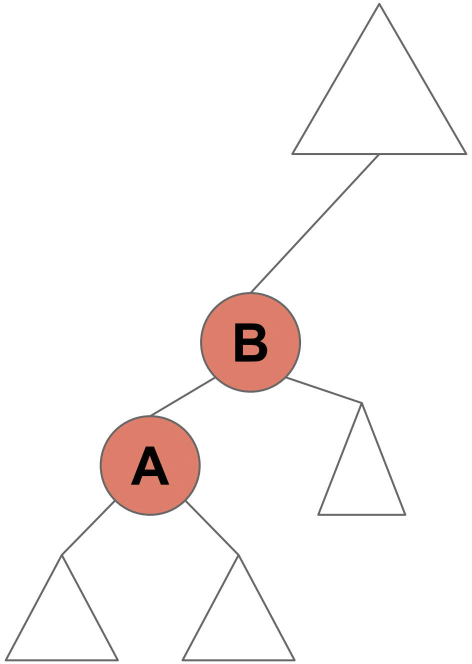


Triangles represent **valid** Red-Black tree fragments

Red-Black Tree

After insertion or deletion, what situations can we encounter?

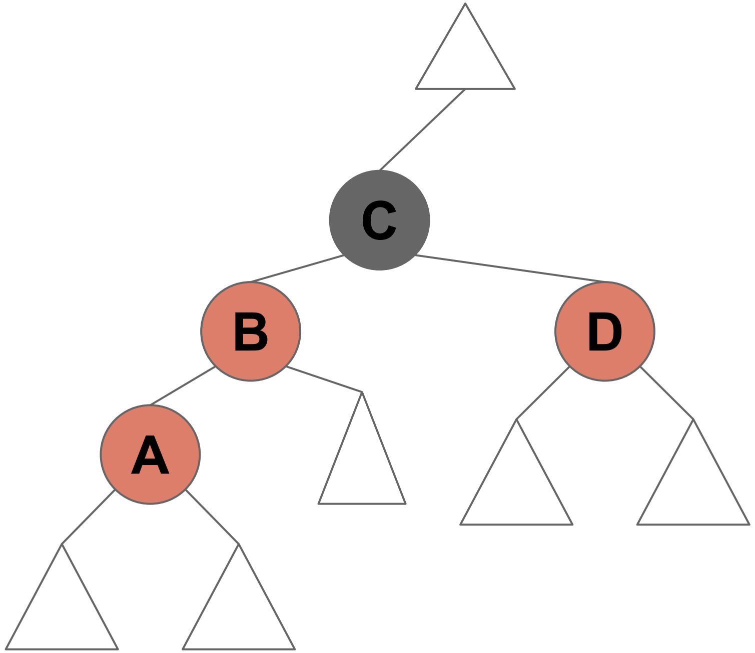
Case 3: The node we are checking is red… and it's parent is red. Now we have to fix the tree.



Red-Black Tree

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…

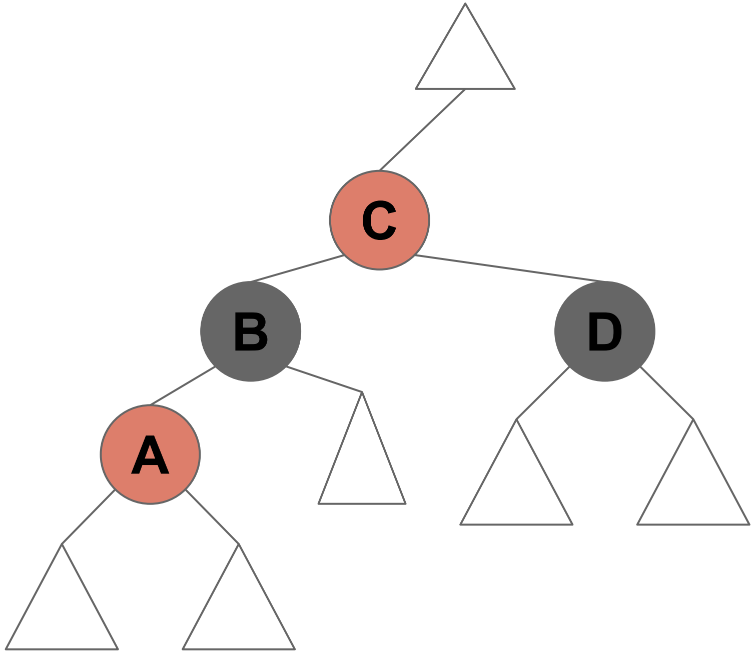


Red-Black Tree

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 Tree

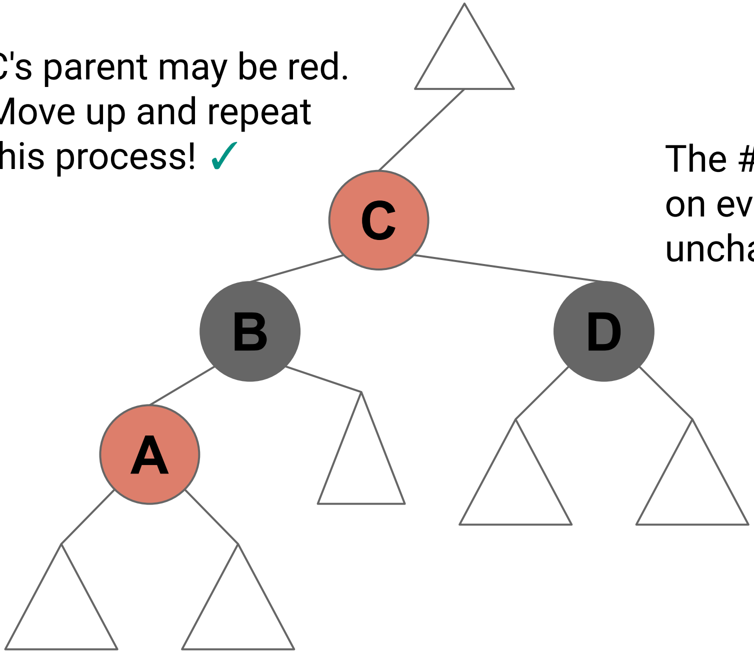
After insertion or deletion, what situations can we encounter?

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?

C's parent may be red. Move up and repeat this process! ✓

The # of black nodes on every path remains unchanged! ✓



Red-Black Tree

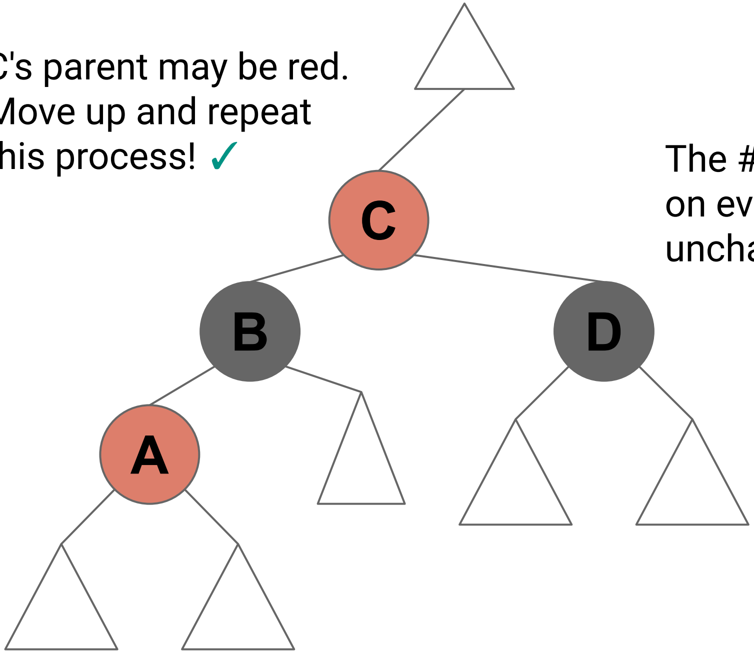
After insertion or deletion, what situations can we encounter?

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

C's parent may be red. Move up and repeat this process! ✓

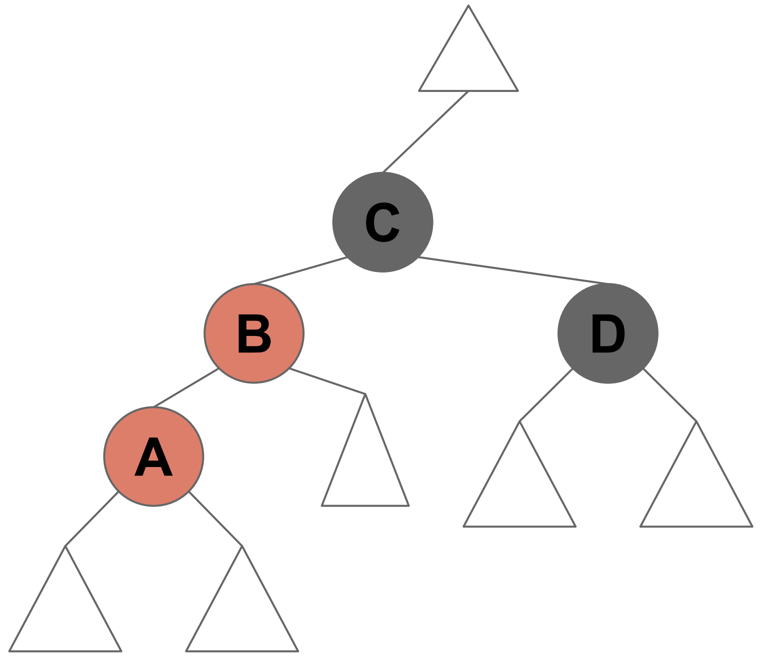


The # of black nodes on every path remains unchanged! ✓

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…

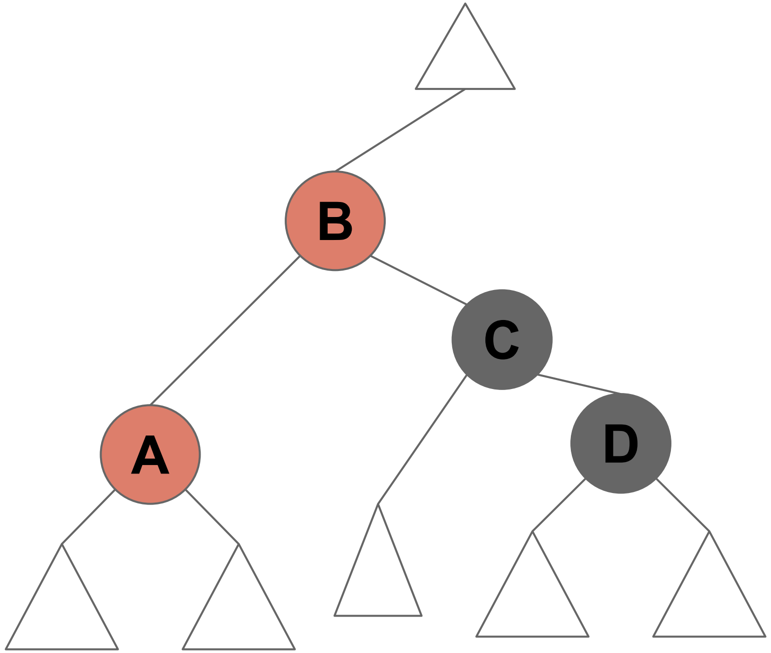


Red-Black Tree

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)

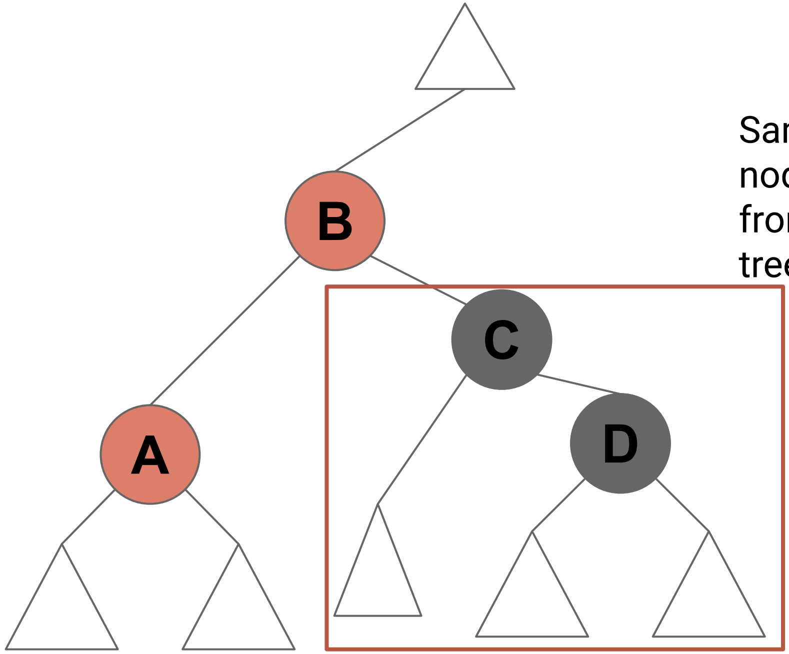


Red-Black Tree

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)



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

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

Red-Black Tree

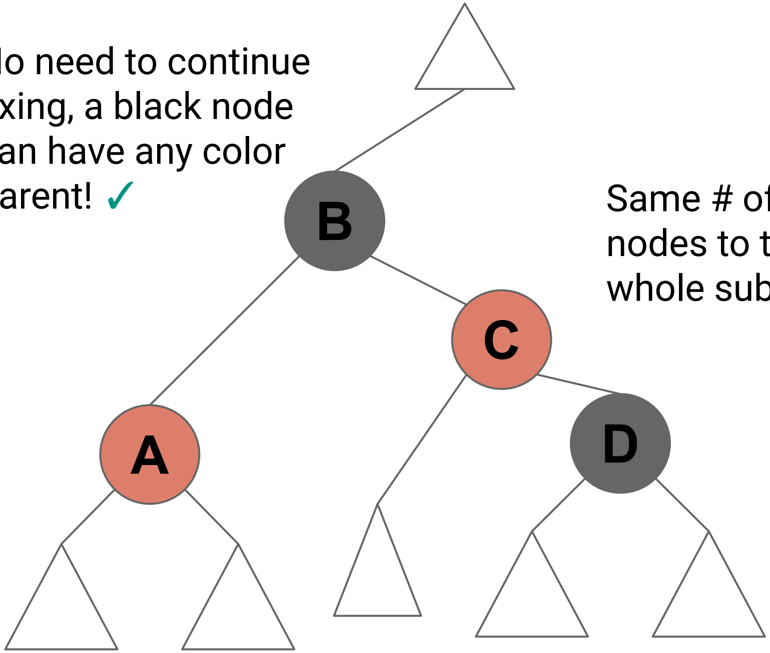
After insertion or deletion, what situations can we encounter?

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)
Recolor(B,C)

No need to continue fixing, a black node can have any color parent! ✓

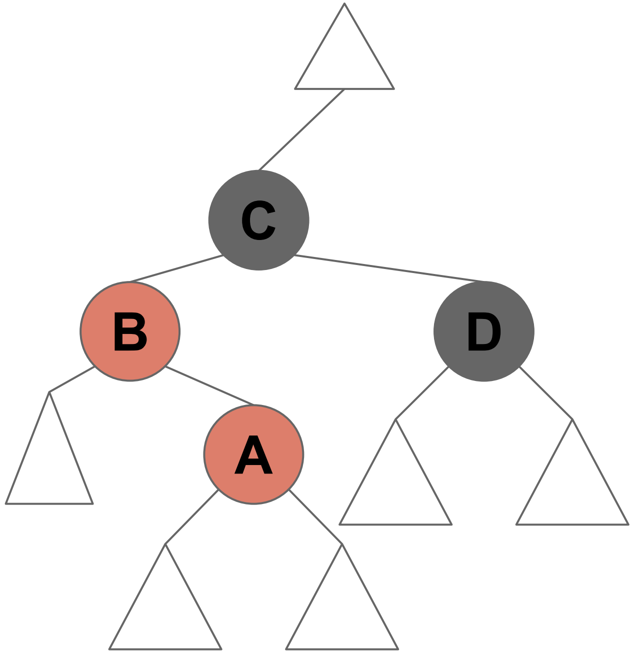
Same # of black nodes to the root for whole subtree! ✓



Red-Black Tree

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

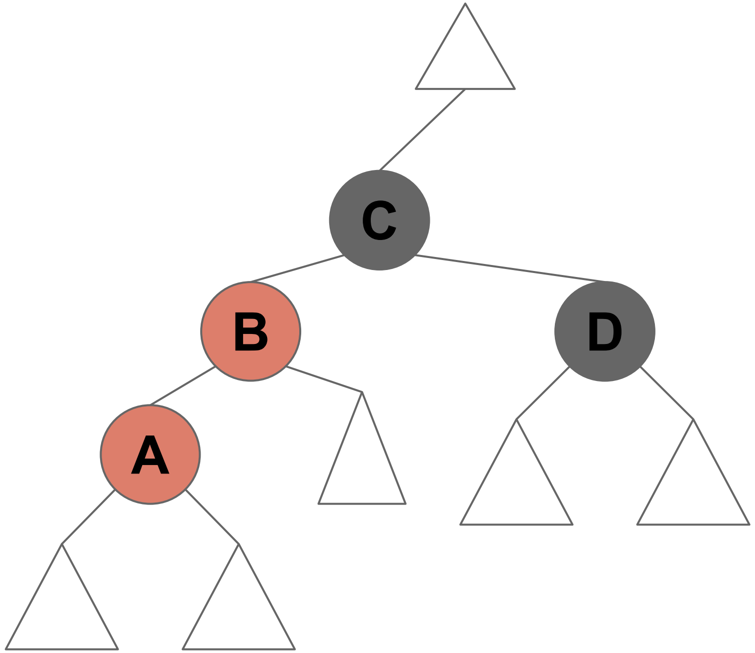


Red-Black Tree

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 Tree

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 $\boldsymbol{O(d)}$ (they involve a constant number of trips from root to leaf at most).

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