

ساختمان داده و الگوریتم ها (CE203)

جلسه هفدهم:
درخت قرمز-سیاه

سجاد شیرعلی شمرضا

پاییز 1400

دوشنبه، 8 آذر 1400

- بخش مرتبط کتاب برای این جلسه: 13



سوال؟

درخت جستجوی متوازن

چگونه درخت جستجو را متوازن نگه داریم؟

SELF-BALANCING SEARCH TREES

There are many different implementations of self-balancing search trees
(e.g. Red-black trees, AVL trees, B-tree, 2-3-4 trees)

SELF-BALANCING SEARCH TREES

There are many different implementations of self-balancing search trees
(e.g. Red-black trees, AVL trees, B-tree, 2-3-4 trees)

Today, we'll go over a key primitive that's used in these implementations:

ROTATIONS

SELF-BALANCING SEARCH TREES

There are many different implementations of self-balancing search trees
(e.g. Red-black trees, AVL trees, B-tree, 2-3-4 trees)

Today, we'll go over a key primitive that's used in these implementations:

ROTATIONS

Note: going forward, we're going to focus on rotations for BINARY search trees (BSTs).

ROTATIONS

IDEA: locally rebalance a node's subtree in $O(1)$ time while maintaining BST property

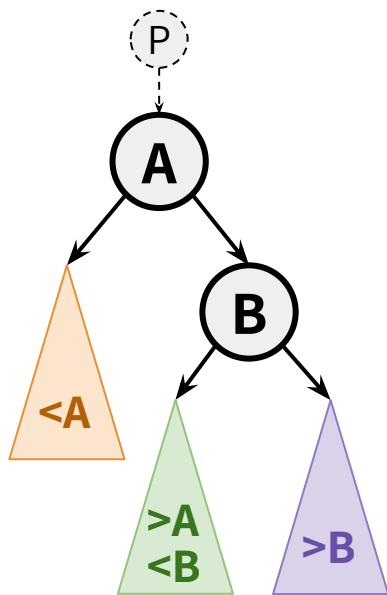
LEFT ROTATION

RIGHT ROTATION

ROTATIONS

IDEA: locally rebalance a node's subtree in $O(1)$ time while maintaining BST property

LEFT ROTATION

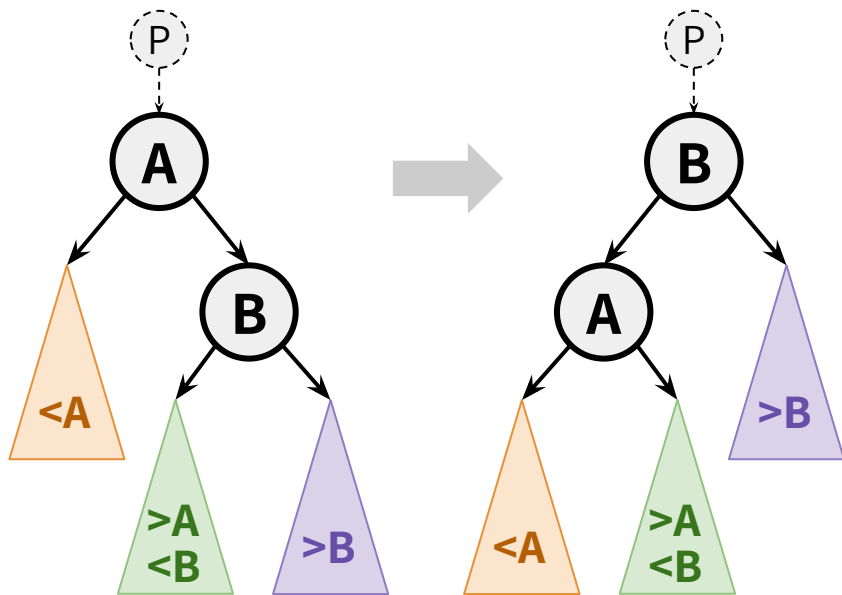


RIGHT ROTATION

ROTATIONS

IDEA: locally rebalance a node's subtree in $O(1)$ time while maintaining BST property

LEFT ROTATION

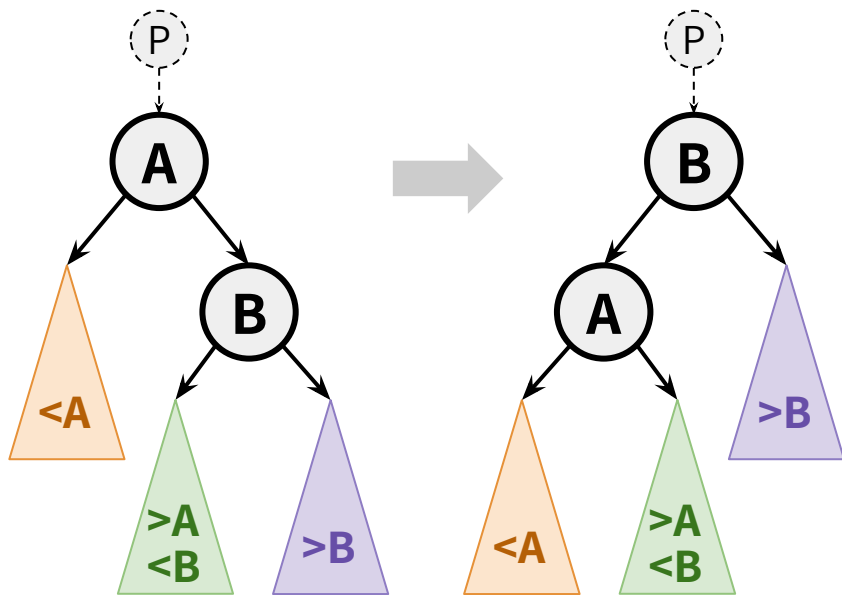


RIGHT ROTATION

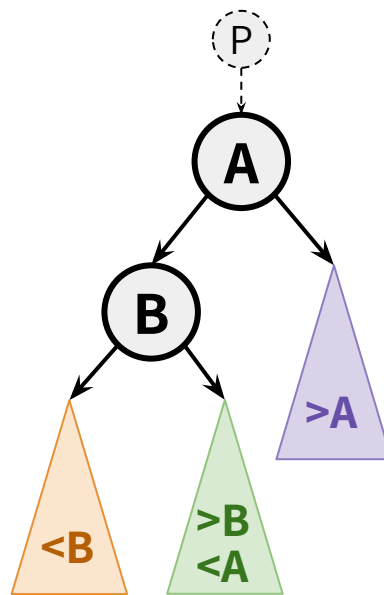
ROTATIONS

IDEA: locally rebalance a node's subtree in $O(1)$ time while maintaining BST property

LEFT ROTATION



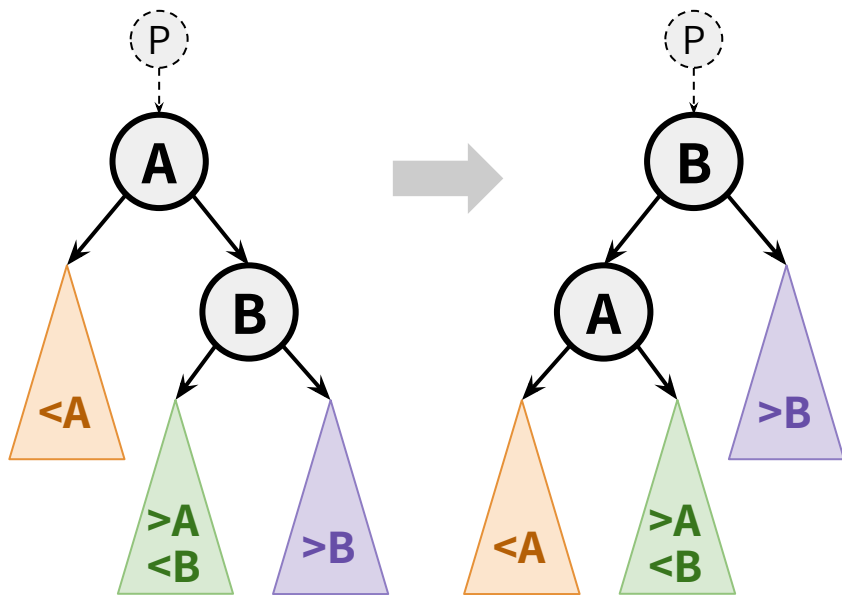
RIGHT ROTATION



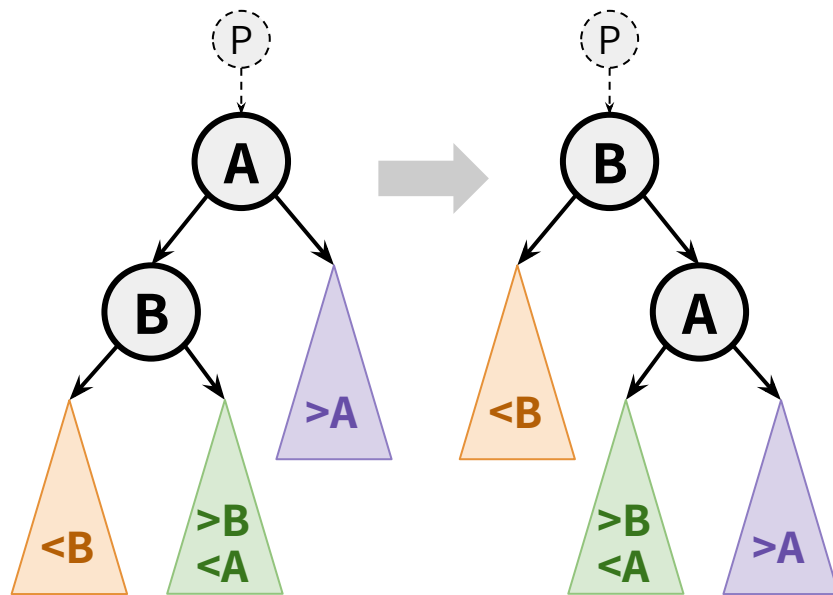
ROTATIONS

IDEA: locally rebalance a node's subtree in $O(1)$ time while maintaining BST property

LEFT ROTATION



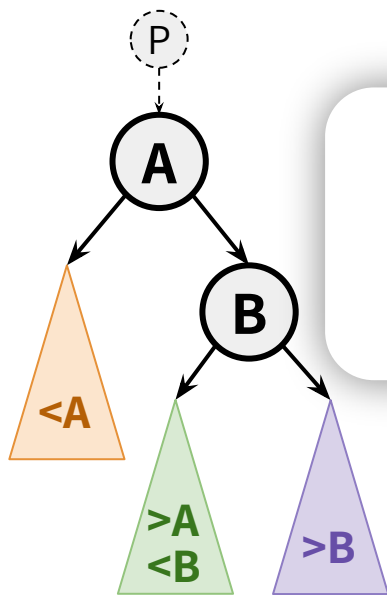
RIGHT ROTATION



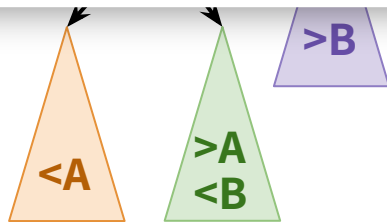
ROTATIONS

IDEA: locally rebalance a node's subtree in $O(1)$ time while maintaining BST property

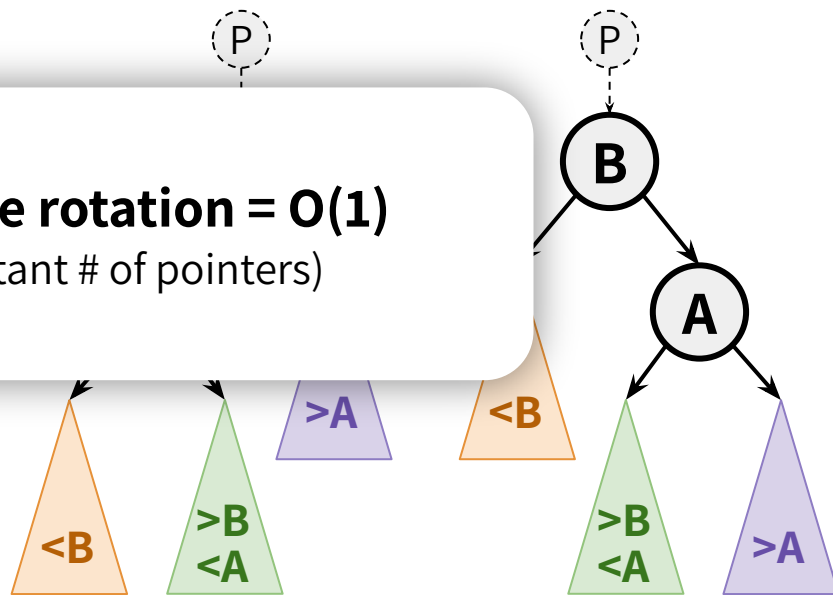
LEFT ROTATION



Runtime of a single rotation = $O(1)$
(only rewires a constant # of pointers)



RIGHT ROTATION





سوال؟

درخت قرمز-سیاه

یک نمونه معروف از درختهای جستجوی متوازن

ROTATIONS

When and how do we apply these rotations?

Let's explore one type of self-balancing BST:

RED-BLACK TREES!

RED-BLACK TREE

A **Red-Black Tree (RB tree)** is a **BST** with the following properties:



RED-BLACK TREE

A **Red-Black Tree (RB tree)** is a **BST** with the following properties:

1. Every node is either **red** or **black**

RED-BLACK TREE

A **Red-Black Tree (RB tree)** is a **BST** with the following properties:

1. Every node is either **red** or **black**
2. The root is a **black** node

RED-BLACK TREE

A **Red-Black Tree (RB tree)** is a **BST** with the following properties:

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child

RED-BLACK TREE

A **Red-Black Tree (RB tree)** is a **BST** with the following properties:

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

RED-BLACK TREE

A **Red-Black Tree (RB tree)** is a **BST** with the following properties:

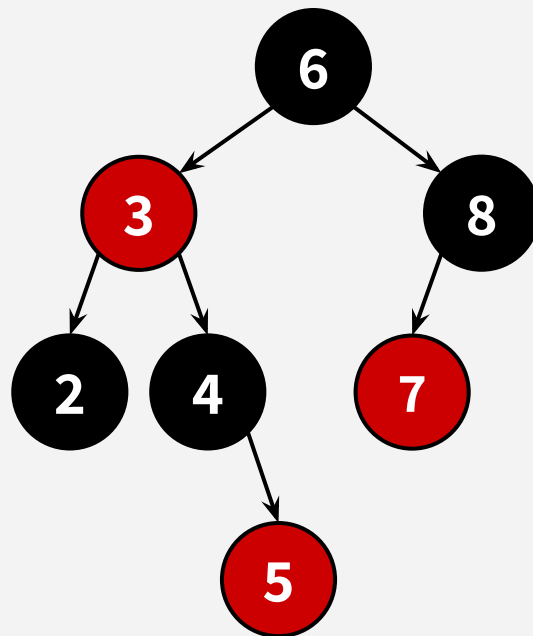
1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

Let's look at some examples & non-examples!

RED-BLACK TREE

A **Red-Black Tree (RB tree)** is a **BST** with the following properties:

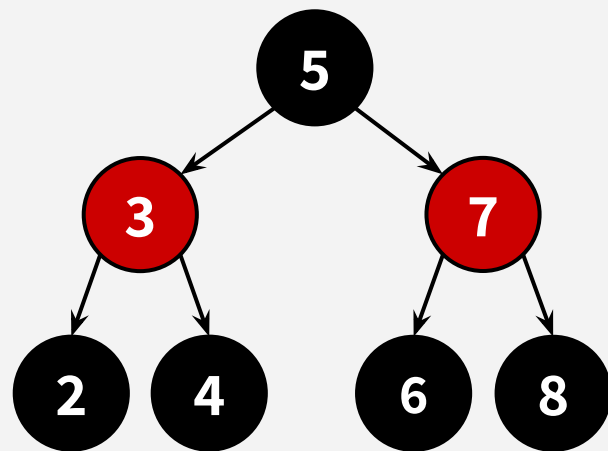
1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them



RED-BLACK TREE

A **Red-Black Tree (RB tree)** is a **BST** with the following properties:

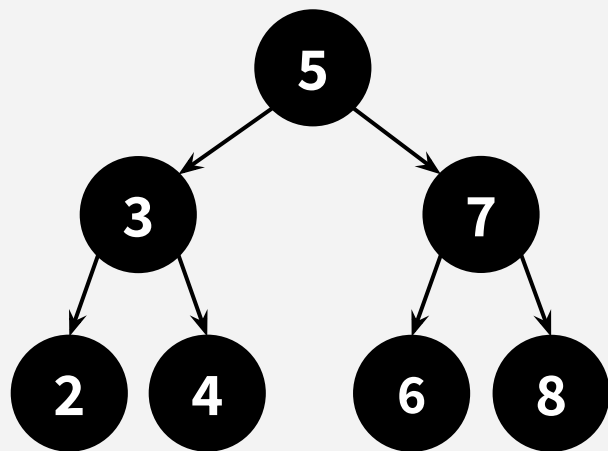
1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them



RED-BLACK TREE

A **Red-Black Tree (RB tree)** is a **BST** with the following properties:

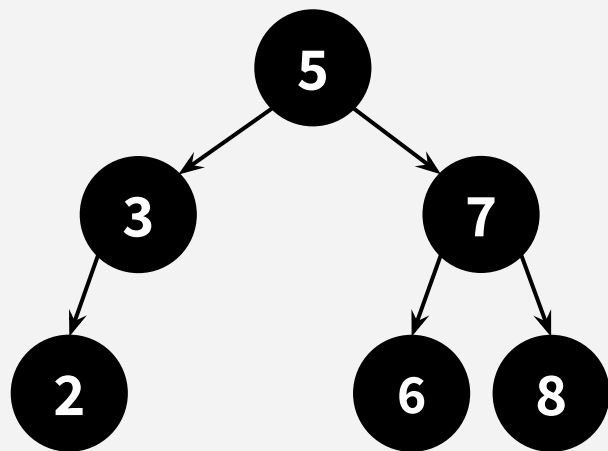
1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them



RED-BLACK TREE

A **Red-Black Tree (RB tree)** is a **BST** with the following properties:

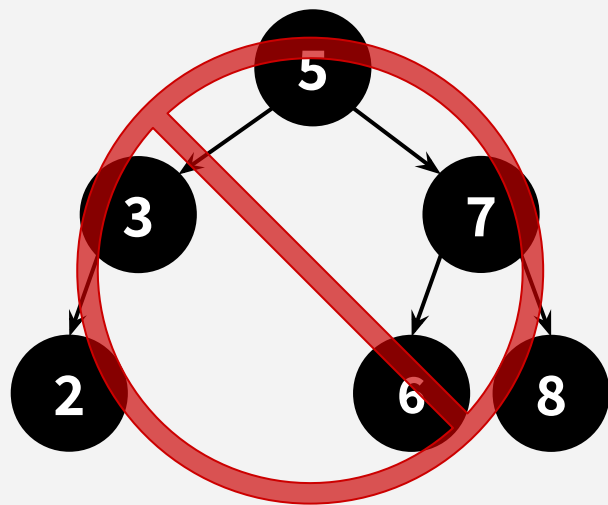
1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them



RED-BLACK TREE

A **Red-Black Tree (RB tree)** is a **BST** with the following properties:

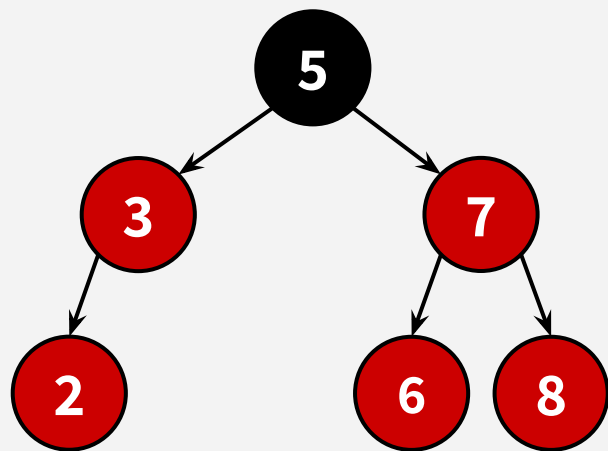
1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them



RED-BLACK TREE

A **Red-Black Tree (RB tree)** is a **BST** with the following properties:

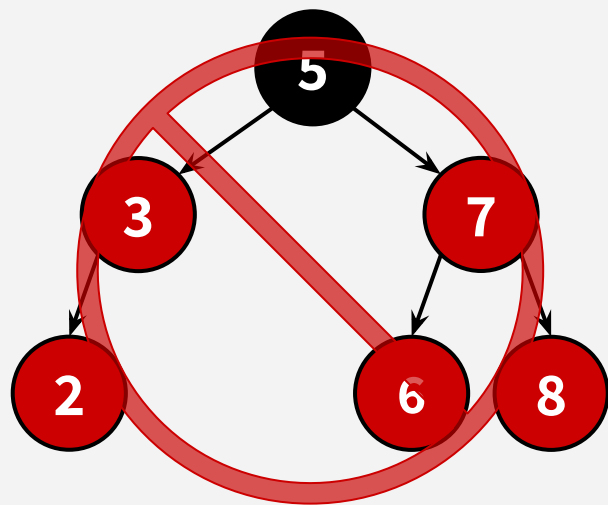
1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them



RED-BLACK TREE

A **Red-Black Tree (RB tree)** is a **BST** with the following properties:

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them





سوال؟

ارتفاع درخت قرمز-سیاه

مزیت ویژگیهای بیان شده برای درخت قرمز-سیاه چیست؟

WHAT'S THE POINT OF THESE RULES?

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

WHAT'S THE POINT OF THESE RULES?

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

Intuitively, these rules are a *proxy* for balance:

The **black** nodes are ~balanced across the tree.

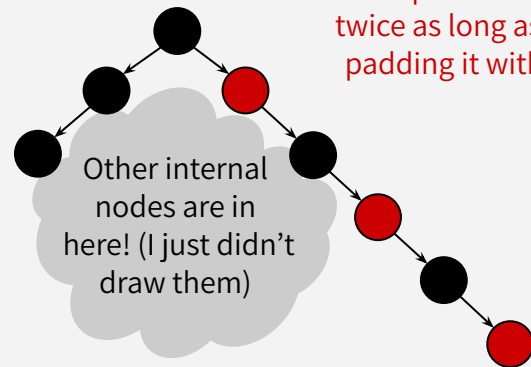
And the **red** nodes might elongate paths but not by much!

WHAT'S THE POINT OF THESE RULES?

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

Intuitively, these rules are a *proxy* for balance:

The **black** nodes are ~balanced across the tree.
And the **red** nodes might elongate paths but not by much!



Rules 3 & 4 guarantee that one path can be at most twice as long as another by padding it with red nodes

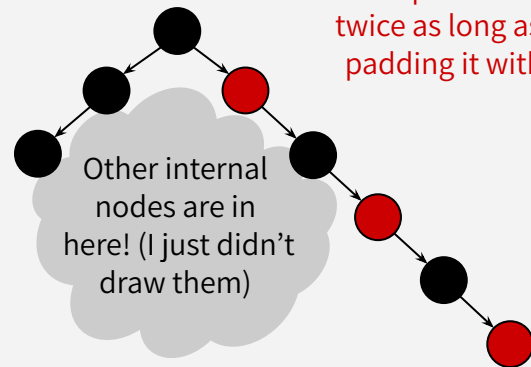
WHAT'S THE POINT OF THESE RULES?

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

Intuitively, these rules are a *proxy* for balance:

The **black** nodes are ~balanced across the tree.
And the **red** nodes might elongate paths but not by much!

More formally...



WHAT'S THE POINT OF THESE RULES?

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

THEOREM: Any Red-Black Tree with **n** nodes has height **$O(\log n)$**

WHAT'S THE POINT OF THESE RULES?

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

THEOREM: Any Red-Black Tree with **n** nodes has height **$O(\log n)$**

PROOF IDEA: We can show that any RB tree with **n** nodes has height $\leq 2 \cdot \log_2(n+1)$

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

(RB TREE PROPERTIES)

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

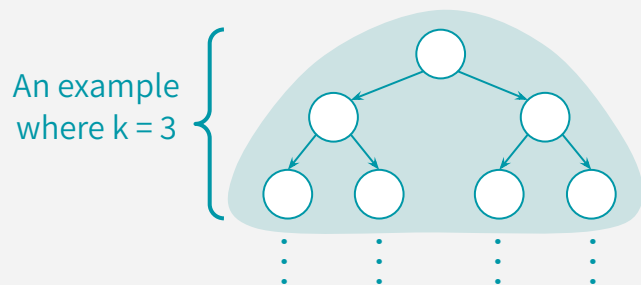
First, suppose every root-NIL path has $\geq k$ nodes.

(RB TREE PROPERTIES)

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

First, suppose every root-NIL path has $\geq k$ nodes. Then the top part of the RB tree must contain a perfectly balanced BST of height $k - 1$.

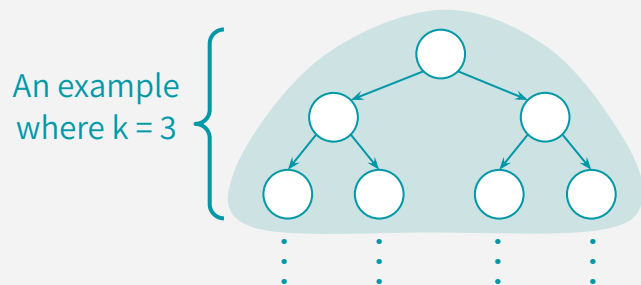


(RB TREE PROPERTIES)

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

First, suppose every root-NIL path has $\geq k$ nodes. Then the top part of the RB tree must contain a perfectly balanced BST of height $k - 1$.



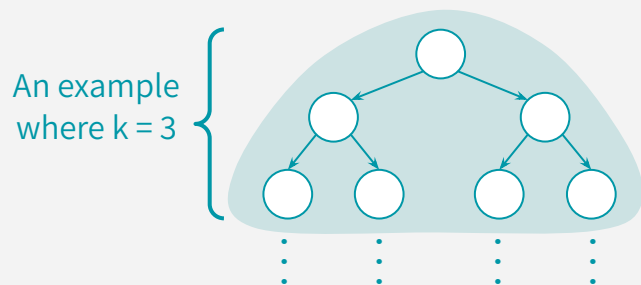
How many nodes are in this blob?

(RB TREE PROPERTIES)

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

First, suppose every root-NIL path has $\geq k$ nodes. Then the top part of the RB tree must contain a perfectly balanced BST of height $k - 1$.



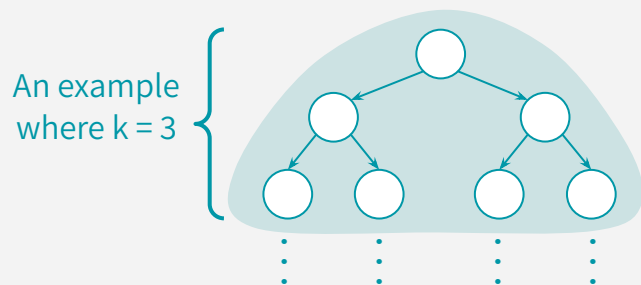
How many nodes are in this blob?
Exactly $2^k - 1$ nodes.

(RB TREE PROPERTIES)

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

First, suppose every root-NIL path has $\geq k$ nodes. Then the top part of the RB tree must contain a perfectly balanced BST of height $k - 1$.



How many nodes are in this blob?
Exactly $2^k - 1$ nodes.

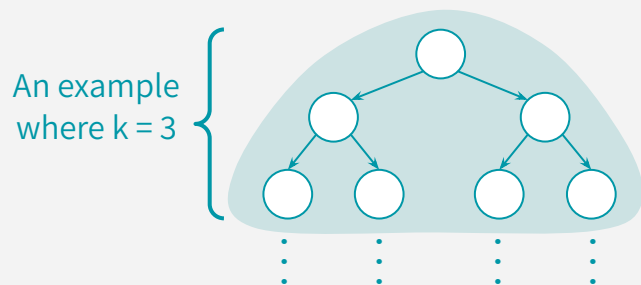
Thus, since there are n nodes in the entire RB tree: $n \geq 2^k - 1$

(RB TREE PROPERTIES)

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

First, suppose every root-NIL path has $\geq k$ nodes. Then the top part of the RB tree must contain a perfectly balanced BST of height $k - 1$.



How many nodes are in this blob?
Exactly $2^k - 1$ nodes.

Thus, since there are n nodes in the entire RB tree: $n \geq 2^k - 1$

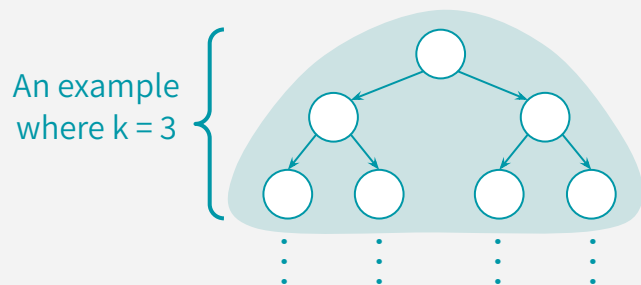
$$\text{i.e. } k \leq \log_2(n+1)$$

(RB TREE PROPERTIES)

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

First, suppose every root-NIL path has $\geq k$ nodes. Then the top part of the RB tree must contain a perfectly balanced BST of height $k - 1$.



How many nodes are in this blob?
Exactly $2^k - 1$ nodes.

Thus, since there are n nodes in the entire RB tree: $n \geq 2^k - 1$

$$\text{i.e. } k \leq \log_2(n+1)$$

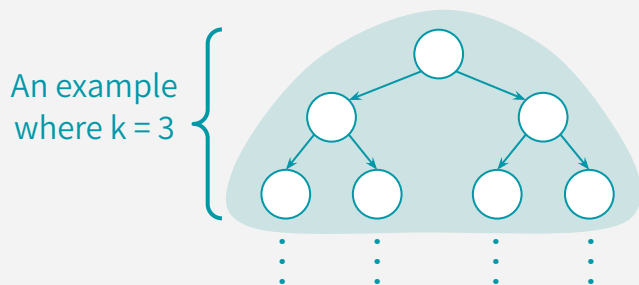
(RB TREE PROPERTIES)

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

This means there must exist some root-NIL path that has $\leq \log_2(n+1)$ nodes on it.

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

First, suppose every root-NIL path has $\geq k$ nodes. Then the top part of the RB tree must contain a perfectly balanced BST of height $k - 1$.



How many nodes are in this blob?
Exactly $2^k - 1$ nodes.

Thus, since there are n nodes in the entire RB tree: $n \geq 2^k - 1$

$$\text{i.e. } k \leq \log_2(n+1)$$

(RB TREE PROPERTIES)

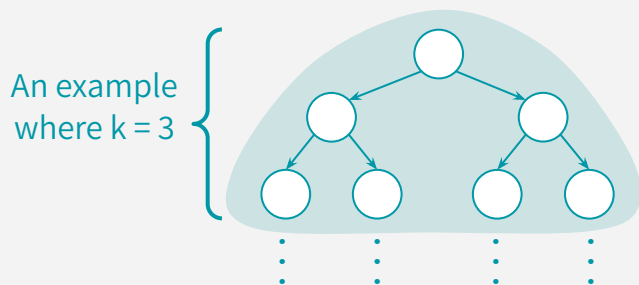
1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

This means there must exist some root-NIL path that has $\leq \log_2(n+1)$ nodes on it.

if all root-NIL paths had $> \log_2(n+1)$ nodes, then k wouldn't be upper bounded by $\log_2(n+1)$

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

First, suppose every root-NIL path has $\geq k$ nodes. Then the top part of the RB tree must contain a perfectly balanced BST of height $k - 1$.



How many nodes are in this blob?
Exactly $2^k - 1$ nodes.

Thus, since there are n nodes in the entire RB tree: $n \geq 2^k - 1$

$$\text{i.e. } k \leq \log_2(n+1)$$

(RB TREE PROPERTIES)

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

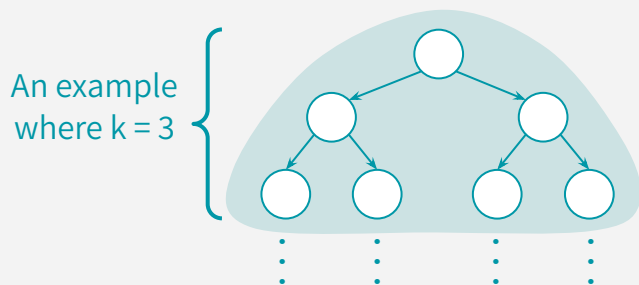
This means there must exist some root-NIL path that has $\leq \log_2(n+1)$ nodes on it.

Consequently, this path must have $\leq \log_2(n+1)$ **black** nodes on it.

if all root-NIL paths had $> \log_2(n+1)$ nodes, then k wouldn't be upper bounded by $\log_2(n+1)$

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

First, suppose every root-NIL path has $\geq k$ nodes. Then the top part of the RB tree must contain a perfectly balanced BST of height $k - 1$.



How many nodes are in this blob?
Exactly $2^k - 1$ nodes.

Thus, since there are n nodes in the entire RB tree: $n \geq 2^k - 1$

$$\text{i.e. } k \leq \log_2(n+1)$$

(RB TREE PROPERTIES)

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

This means there must exist some root-NIL path that has $\leq \log_2(n+1)$ nodes on it.

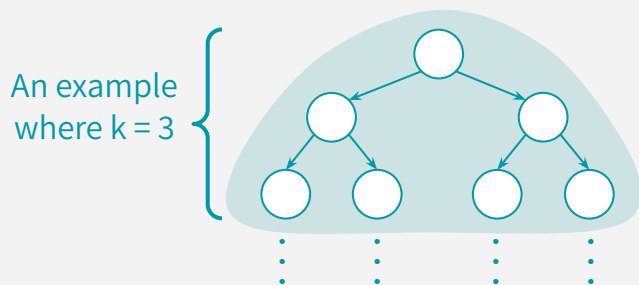
Consequently, this path must have $\leq \log_2(n+1)$ **black** nodes on it.

By PROPERTY 4: every root-NIL path has $\leq \log_2(n+1)$ **black** nodes on it.

if all root-NIL paths had $> \log_2(n+1)$ nodes, then k wouldn't be upper bounded by $\log_2(n+1)$

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

First, suppose every root-NIL path has $\geq k$ nodes. Then the top part of the RB tree must contain a perfectly balanced BST of height $k - 1$.



How many nodes are in this blob?
Exactly $2^k - 1$ nodes.

Thus, since there are n nodes in the entire RB tree: $n \geq 2^k - 1$

$$\text{i.e. } k \leq \log_2(n+1)$$

(RB TREE PROPERTIES)

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

This means there must exist some root-NIL path that has $\leq \log_2(n+1)$ nodes on it.

Consequently, this path must have $\leq \log_2(n+1)$ **black** nodes on it.

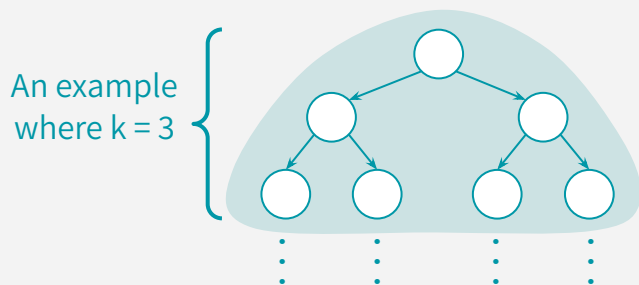
By PROPERTY 4: every root-NIL path has $\leq \log_2(n+1)$ **black** nodes on it.

By PROPERTY 3: every root-NIL path has $\leq 2 \cdot \log_2(n+1)$ total nodes on it.

if all root-NIL paths had $> \log_2(n+1)$ nodes, then k wouldn't be upper bounded by $\log_2(n+1)$

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

First, suppose every root-NIL path has $\geq k$ nodes. Then the top part of the RB tree must contain a perfectly balanced BST of height $k - 1$.



How many nodes are in this blob?
Exactly $2^k - 1$ nodes.

Thus, since there are n nodes in the entire RB tree: $n \geq 2^k - 1$

$$\text{i.e. } k \leq \log_2(n+1)$$

(RB TREE PROPERTIES)

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

This means there must exist some root-NIL path that has $\leq \log_2(n+1)$ nodes on it.

Consequently, this path must have $\leq \log_2(n+1)$ **black** nodes on it.

By PROPERTY 4: every root-NIL path has $\leq \log_2(n+1)$ **black** nodes on it.

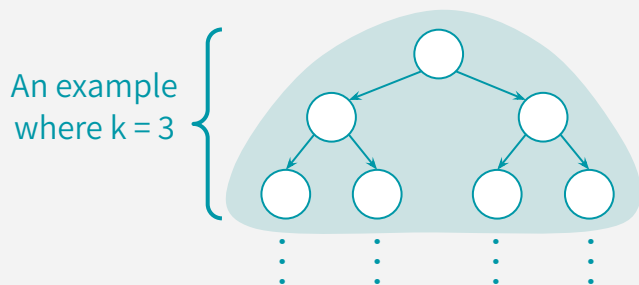
By PROPERTY 3: every root-NIL path has $\leq 2 \cdot \log_2(n+1)$ total nodes on it.

if all root-NIL paths had $> \log_2(n+1)$ nodes, then k wouldn't be upper bounded by $\log_2(n+1)$

Red and black nodes must alternate, so the # of red nodes on the path is at most the # of black nodes

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

First, suppose every root-NIL path has $\geq k$ nodes. Then the top part of the RB tree must contain a perfectly balanced BST of height $k - 1$.



How many nodes are in this blob?
Exactly $2^k - 1$ nodes.

Thus, since there are n nodes in the entire RB tree: $n \geq 2^k - 1$

$$\text{i.e. } k \leq \log_2(n+1)$$

(RB TREE PROPERTIES)

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

This means there must exist some root-NIL path that has $\leq \log_2(n+1)$ nodes on it.

Consequently, this path must have $\leq \log_2(n+1)$ **black** nodes on it.

By PROPERTY 4: every root-NIL path has $\leq \log_2(n+1)$ **black** nodes on it.

By PROPERTY 3: every root-NIL path has $\leq 2 \cdot \log_2(n+1)$ total nodes on it.

Thus, the height of the RB tree is at most $2 \cdot \log_2(n+1)$, aka **the height of any RB tree is $O(\log n)$** .

if all root-NIL paths had $> \log_2(n+1)$ nodes, then k wouldn't be upper bounded by $\log_2(n+1)$

Red and black nodes must alternate, so the # of red nodes on the path is at most the # of black nodes

$O(\log n)$ HEIGHT GUARANTEE (PROOF)

First, suppose every root-NIL path has $\geq k$ nodes. Then the top part of the RB tree is a path of $\geq k$ nodes. But by (1) and (2), the root is black and its children are red. So the path has at least 2 nodes. So $k \leq 2$.

(RB TREE PROPERTIES)

There's a lot going on, so here's how you should assess your understanding:

**Properties 3 and 4 are the non-trivial rules.
Their purpose should ~intuitively make sense.**

This

Con

By PROPERTY 4: every root-NIL path has $\leq \log_2(n+1)$ black nodes on it.

By PROPERTY 3: every root-NIL path has $\leq 2 \cdot \log_2(n+1)$ total nodes on it.

red and black nodes must alternate,
so the # of red nodes on the path is
at most the # of black nodes

Thus, the height of the RB tree is at most $2 \cdot \log_2(n+1)$, aka **the height of any RB tree is $O(\log n)$.**



سوال؟

تغییر درخت قرمز-سیاه

چگونگی اضافه و حذف کردن در درخت قرمز-سیاه

WHAT HAVE WE LEARNED?

The height of an RB Tree is $O(\log n)$.

Runtime of **SEARCH** in an RB Tree = **$O(\text{height})$**
= $O(\log n)$

WHAT HAVE WE LEARNED?

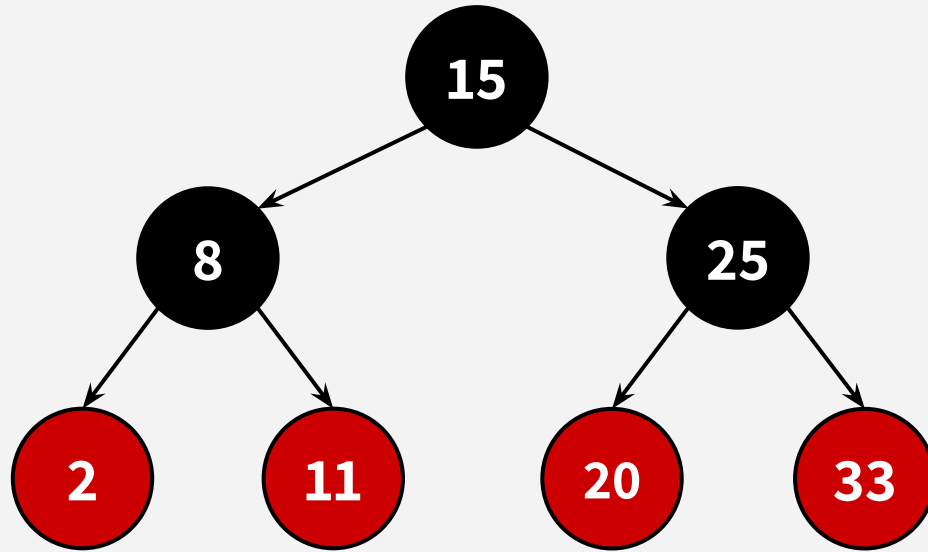
The height of an RB Tree is $O(\log n)$.

Runtime of **SEARCH** in an RB Tree = **$O(\text{height})$**
= $O(\log n)$

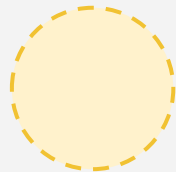
What about INSERT/DELETE?

These are the two operations that actually modify the RB Tree, so we need to make sure that we insert & delete without violating our precious RB Tree properties...

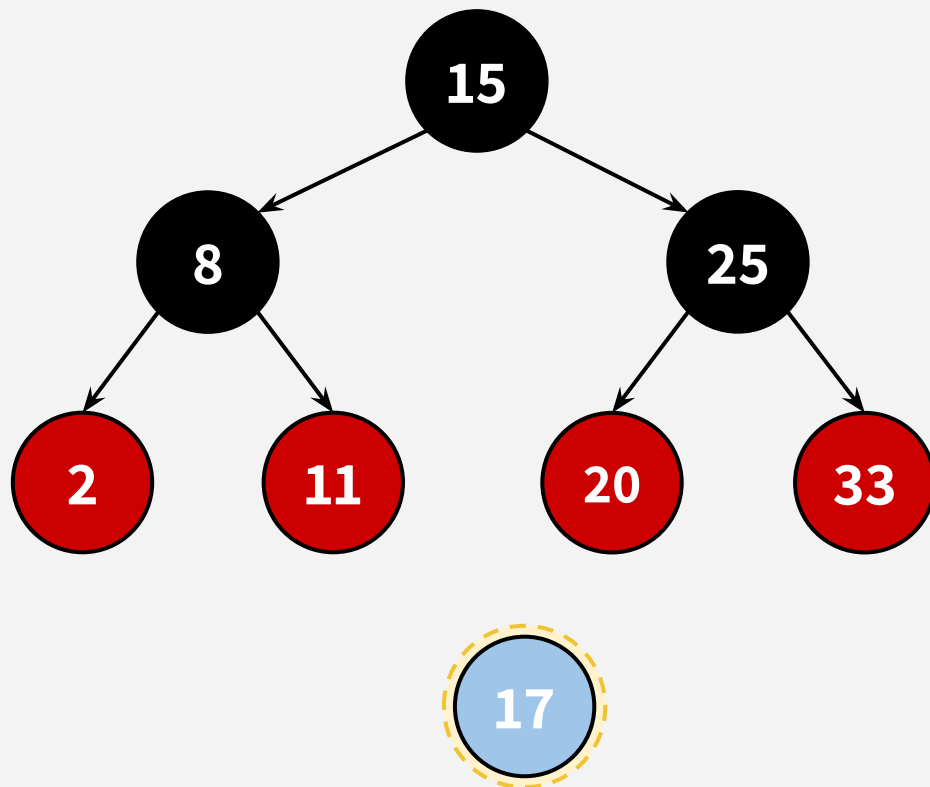
INSERTING IN AN RB TREE



EXAMPLE: Insert 17.



INSERTING IN AN RB TREE



EXAMPLE: Insert 17.

What do we do with 17?

Do we color it **red**?

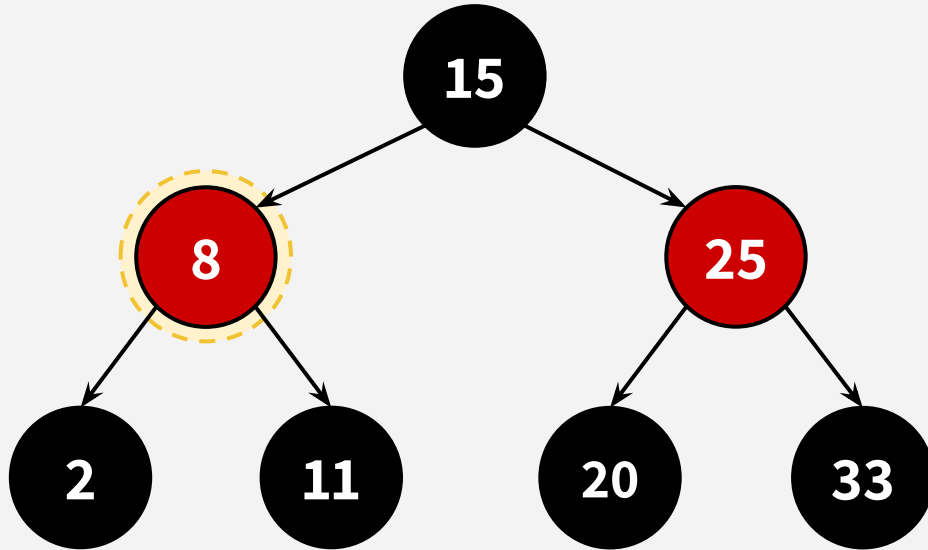
Do we color it **black**?

Do we need to change the
color of other nodes?

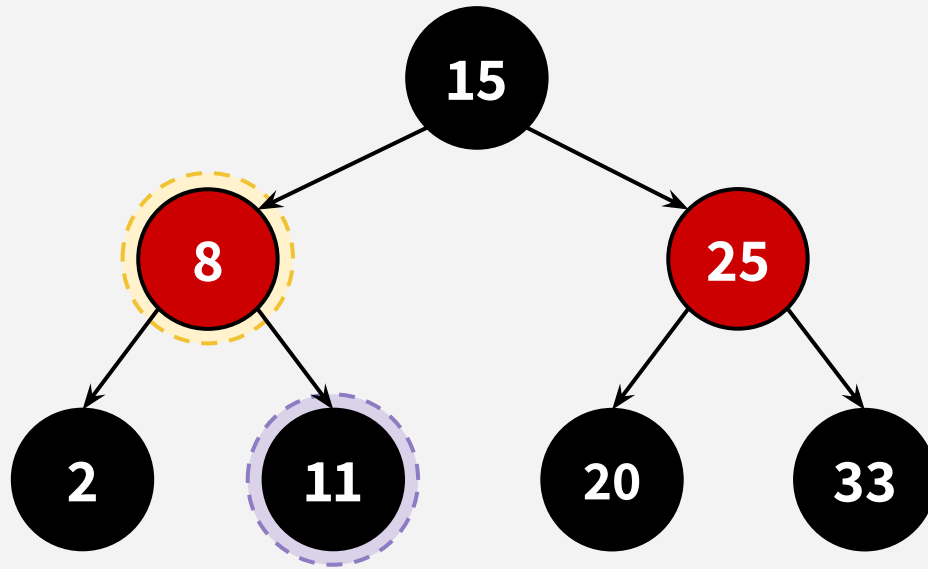
What if we insert 16 next?

DELETING FROM AN RB TREE

EXAMPLE: Delete 8.

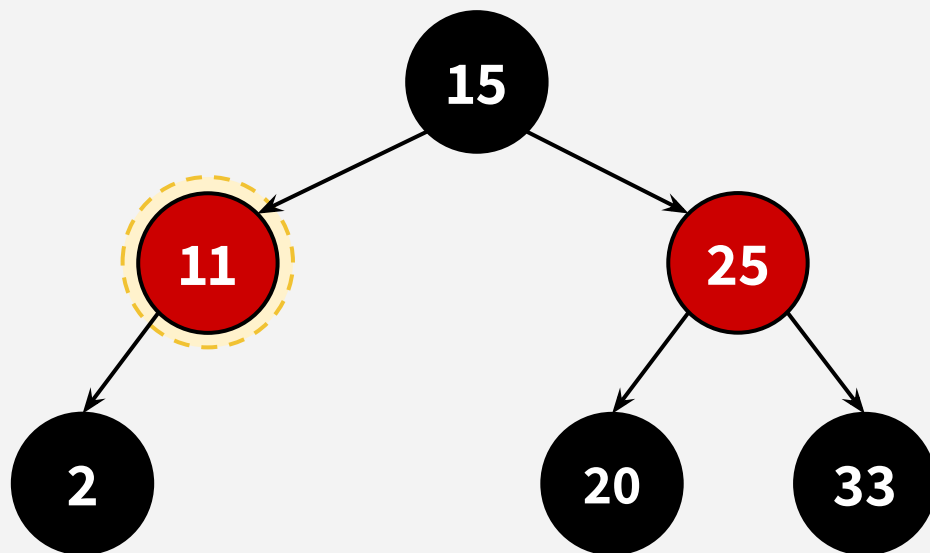


DELETING FROM AN RB TREE



EXAMPLE: Delete 8.
(replace with immediate successor)

DELETING FROM AN RB TREE



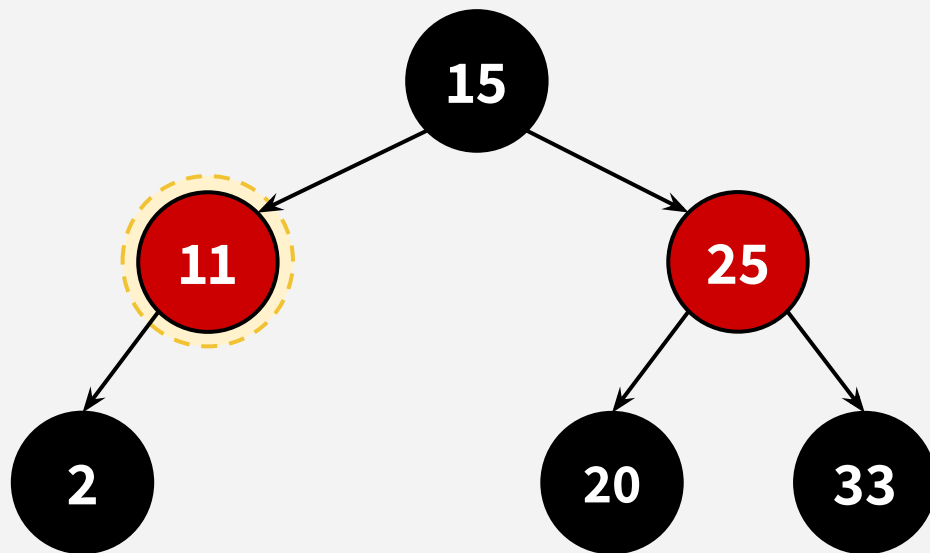
EXAMPLE: Delete 8.
(replace with immediate successor)

**Now we've violated
Property 4!**

(all root-NIL paths must have
the same # of black nodes)

How do we fix this up?

DELETING FROM AN RB TREE



EXAMPLE: Delete 8.
(replace with immediate successor)

**Now we've violated
Property 4!**

(all root-NIL paths must have
the same # of black nodes)

How do we fix this up?

Fixing up deletions is *complicated*. See CLRS Section 13.4 if you're curious!



سوال؟

درج در درخت قرمز-سیاه

چگونگی متوازن نگه داشتن درخت هنگام درج

INSERT IN RB TREES

High-level plan

Insert as normal (same insert as BST), and then fix.

Fix = recolor and/or apply rotations until RB Tree properties are met.

INSERT IN RB TREES

High-level plan

Insert as normal (same insert as BST), and then fix.

Fix = recolor and/or apply rotations until RB Tree properties are met.

INSERT(x):

- Insert **x** normally (**x** becomes a leaf)
- Color **x** **red**

INSERT IN RB TREES

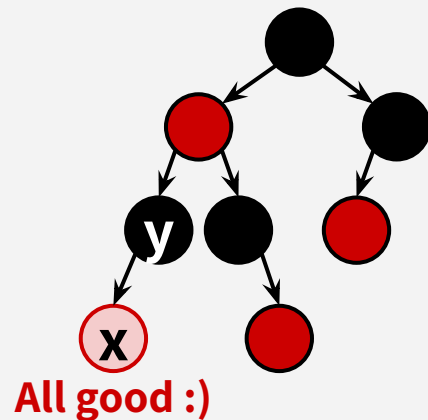
High-level plan

Insert as normal (same insert as BST), and then fix.

Fix = recolor and/or apply rotations until RB Tree properties are met.

INSERT(**x**):

- Insert **x** normally (**x** becomes a leaf)
- Color **x** **red**
- If **x**'s parent **y** is **black**, then we're done!



INSERT IN RB TREES

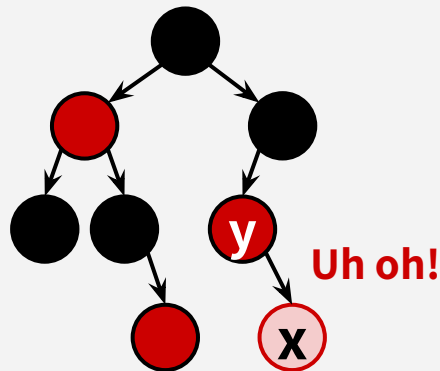
High-level plan

Insert as normal (same insert as BST), and then fix.

Fix = recolor and/or apply rotations until RB Tree properties are met.

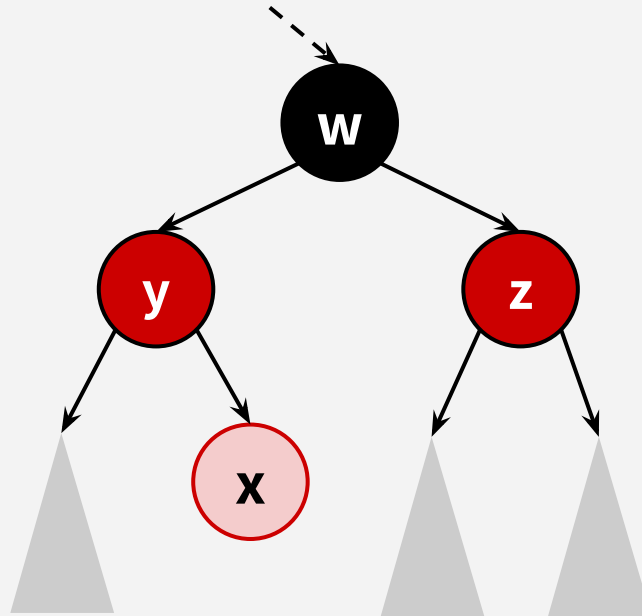
INSERT(**x**):

- Insert **x** normally (**x** becomes a leaf)
- Color **x** **red**
- If **x**'s parent **y** is **black**, then we're done!
- Otherwise, **y** is **red**, so we have two red nodes in a row and need to do some fixing!



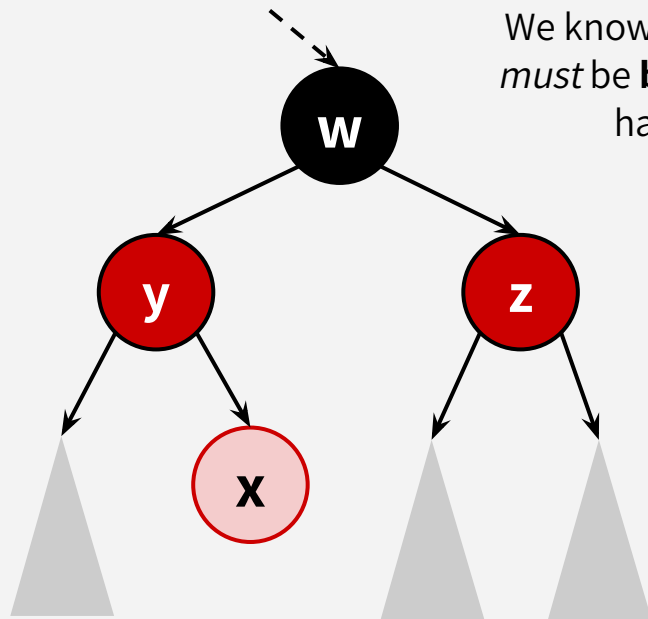
FIXING THINGS UP: CASE 1

CASE 1: parent **y** is **red**, and “uncle” **z** is **red** too!



FIXING THINGS UP: CASE 1

CASE 1: parent **y** is **red**, and “uncle” **z** is **red** too!



We know **w** (x's grandparent & y's parent) *must* be **black** because we could not have had two **red** nodes in a row.

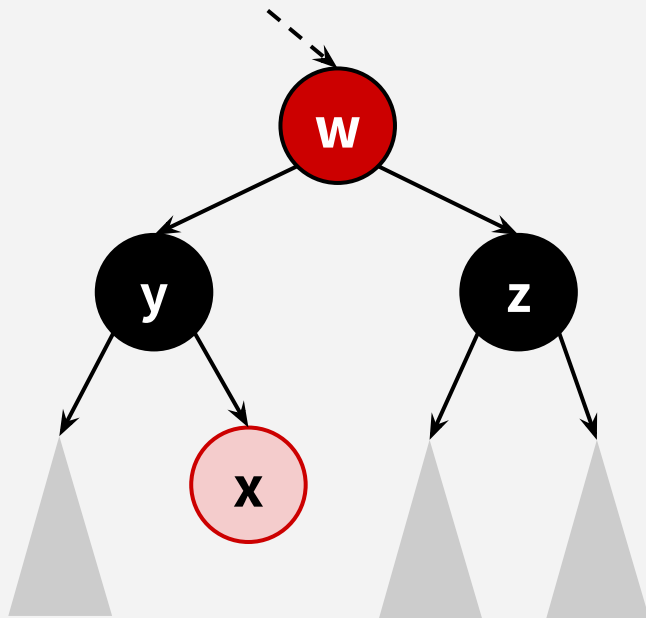
FIXING THINGS UP: CASE 1

CASE 1: parent **y** is **red**, and “uncle” **z** is **red** too!

Recolor!

Change **w** to **red** &
change **y** and **z** both to **black**

One recolor = $O(1)$ time



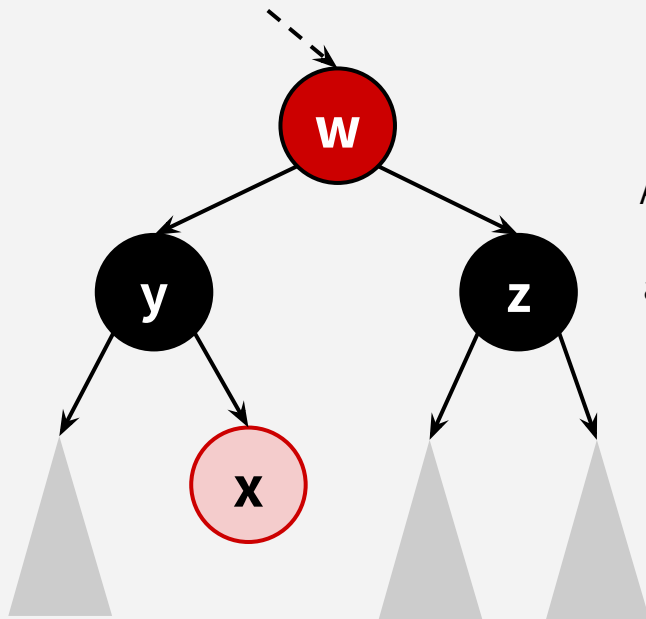
FIXING THINGS UP: CASE 1

CASE 1: parent **y** is **red**, and “uncle” **z** is **red** too!

Recolor!

Change **w** to **red** &
change **y** and **z** both to **black**

One recolor = $O(1)$ time



This doesn't hurt Property 4!

All root-NIL paths that interact with **w**, **y**, or **z**, all have to go through **w** and would hit exactly one of **y** or **z**.

Before, **w** contributed one **black** node to each of those paths, and now, **y** (or **z**) still contributes one **black** node (instead of **w**).

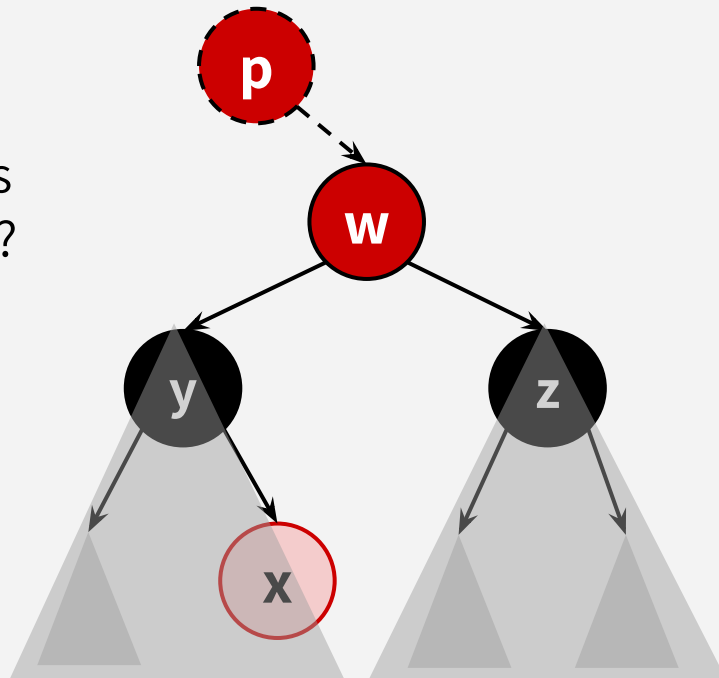


سوال؟

FIXING THINGS UP: CASE 1

CASE 1: parent **y** is **red**, and “uncle” **z** is **red** too!

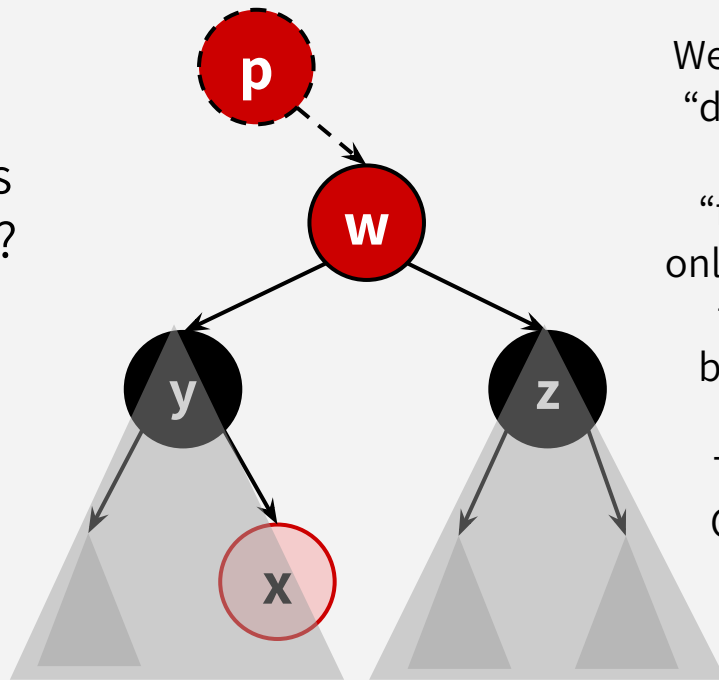
But wait! What if **w**'s parent was also **red**?



FIXING THINGS UP: CASE 1

CASE 1: parent **y** is **red**, and “uncle” **z** is **red** too!

But wait! What if **w**'s parent was also **red**?



We basically just propagated the “double-red” violation upward!

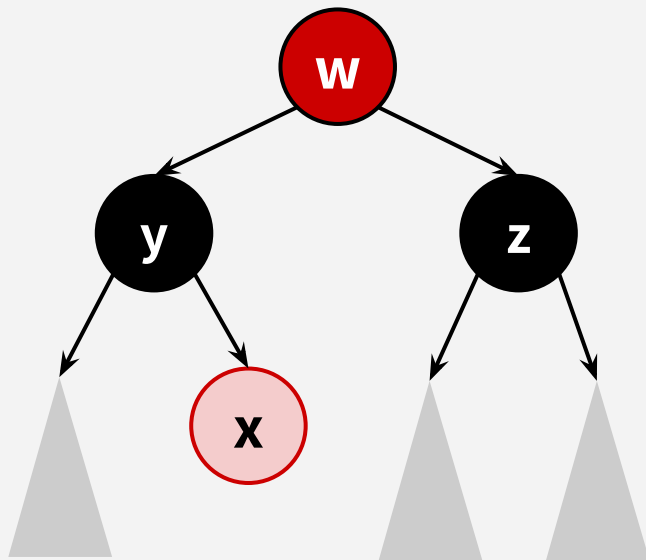
We can recursively do this “fix-up”. This propagation can only happen $O(\log n)$ times, since the tree was a valid RB Tree before this INSERT operation!

Thus, overall, INSERT in this CASE would be $O(\log n)$ still.

FIXING THINGS UP: CASE 1

CASE 1: parent **y** is **red**, and “uncle” **z** is **red** too!

But wait again!!
What if **w** is the root?
The root can't be **red**!



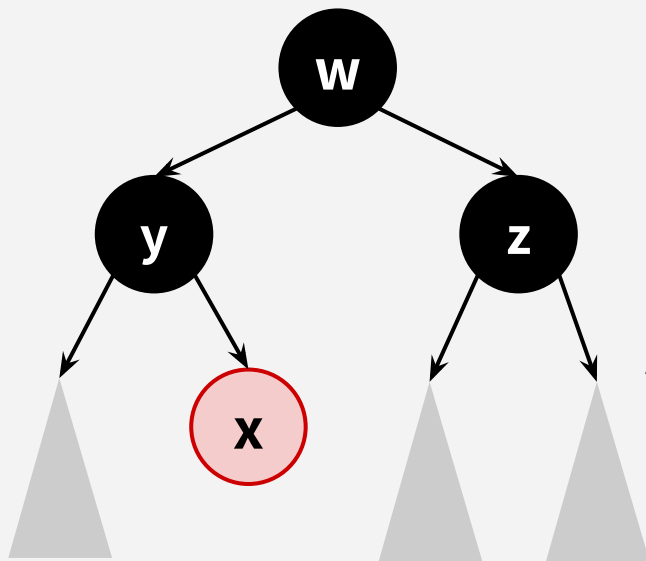


سوال؟

FIXING THINGS UP: CASE 1

CASE 1: parent **y** is **red**, and “uncle” **z** is **red** too!

But wait again!!
What if **w** is the root?
The root can't be **red**!



No stress at all,
just color it **black**!

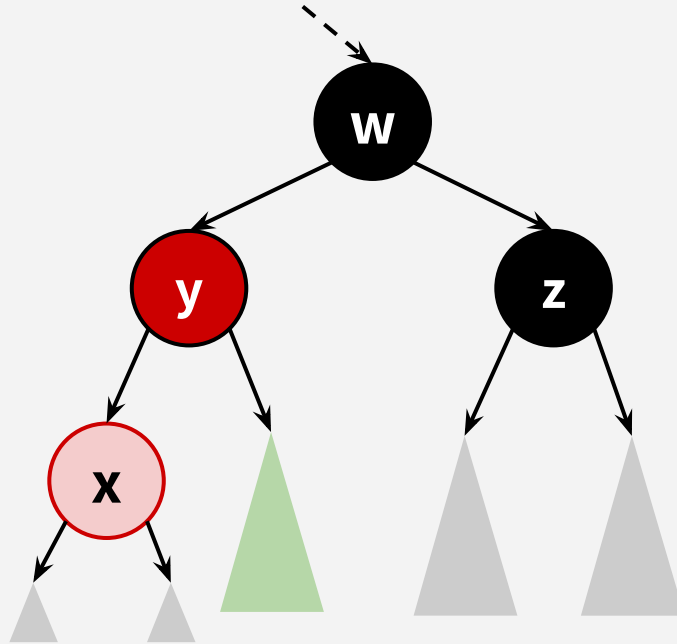
If **w** is the root, then **w** appears
once on every root-NIL path.
Thus, changing **w** to **black** will
just add 1 to every root-NIL path
and Property 4 is still preserved!



سوال؟

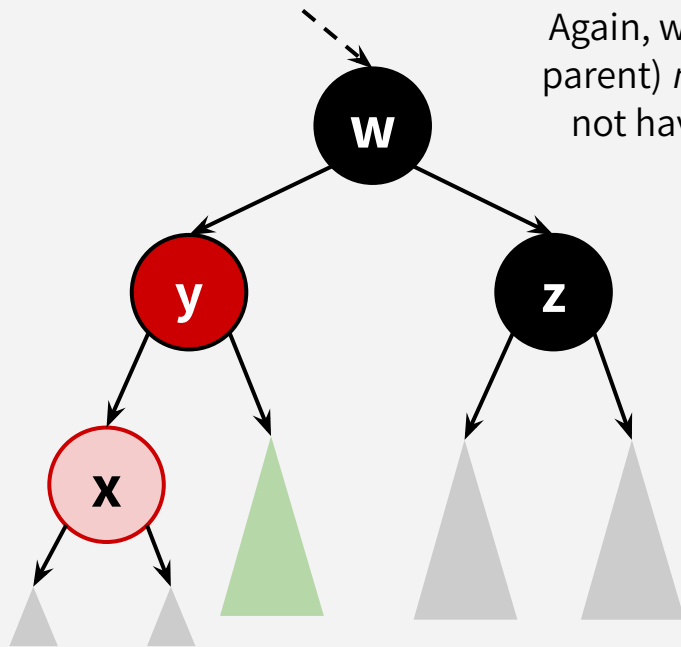
FIXING THINGS UP: CASE 2

CASE 2: parent **y** is **red**, and “uncle” **z** is **black** (or NIL)!



FIXING THINGS UP: CASE 2

CASE 2: parent **y** is **red**, and “uncle” **z** is **black** (or NIL)!



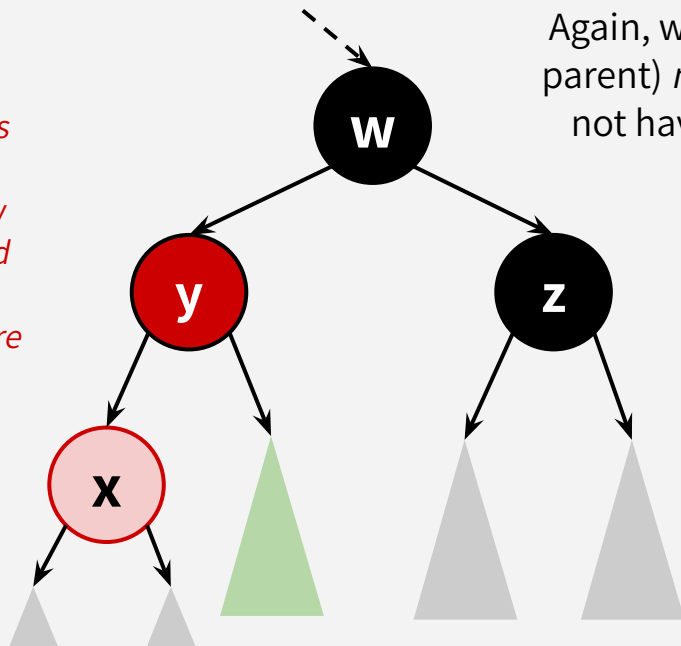
Again, we know **w** (x's grandparent & y's parent) *must* be **black** because we could not have had two **red** nodes in a row.

FIXING THINGS UP: CASE 2

CASE 2: parent **y** is **red**, and “uncle” **z** is **black** (or NIL)!

DISCLAIMER:

*This is just one of several sub-cases that fall under this Case 2. To understand all cases and why/how they work **intuitively**, you can read about 2-3-4 trees, which are an **isometry** of RB Trees. 2-3-4 trees are much easier to understand, and operations on 2-3-4 trees map to rotation routines for RB Trees!*



Again, we know **w** (x's grandparent & y's parent) *must* be **black** because we could not have had two **red** nodes in a row.

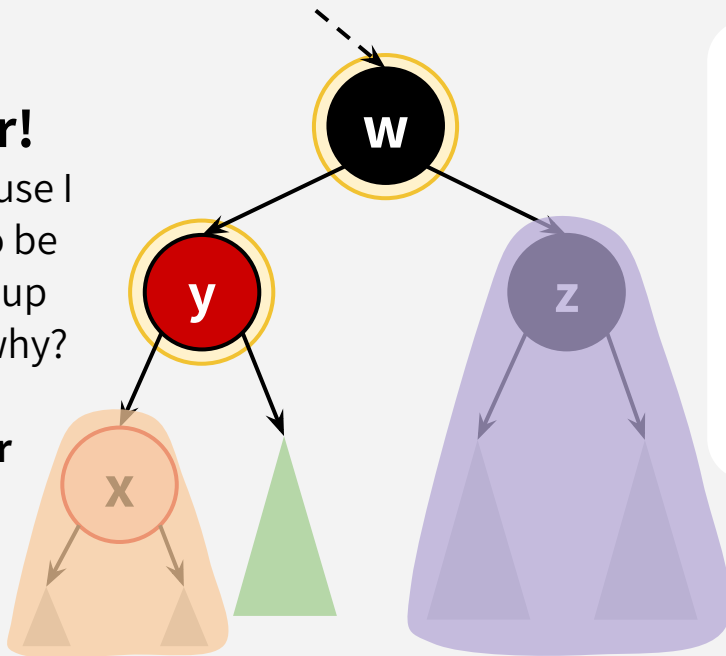
FIXING THINGS UP: CASE 2

CASE 2: parent **y** is **red**, and “uncle” **z** is **black** (or NIL)!

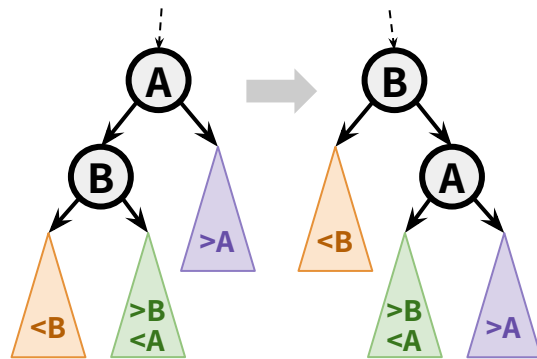
ROTATE & Recolor!

I need to rotate first, because I can't just recolor **x** or **y** to be **black**! That would mess up Property 4 - can you see why?

One rotation + recolor
= $O(1)$ time



RIGHT ROTATION



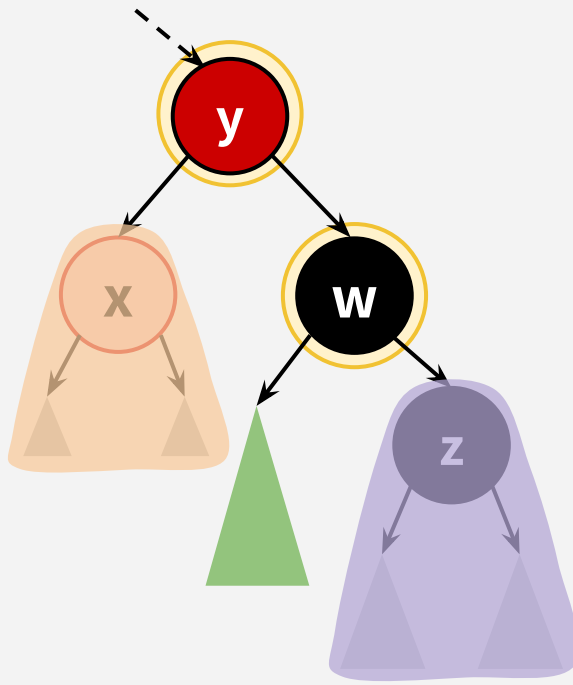
FIXING THINGS UP: CASE 2

CASE 2: parent **y** is **red**, and “uncle” **z** is **black** (or NIL)!

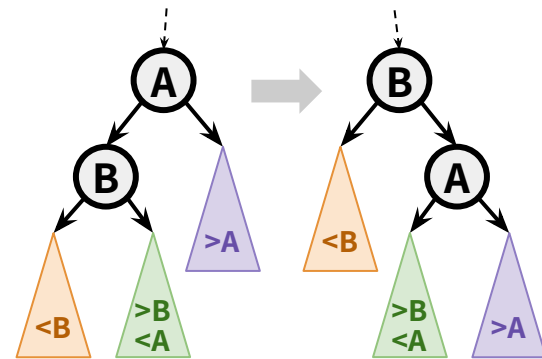
ROTATE & Recolor!

I need to rotate first, because I can't just recolor **x** or **y** to be **black**! That would mess up Property 4 - can you see why?

One rotation + recolor
= $O(1)$ time



RIGHT ROTATION



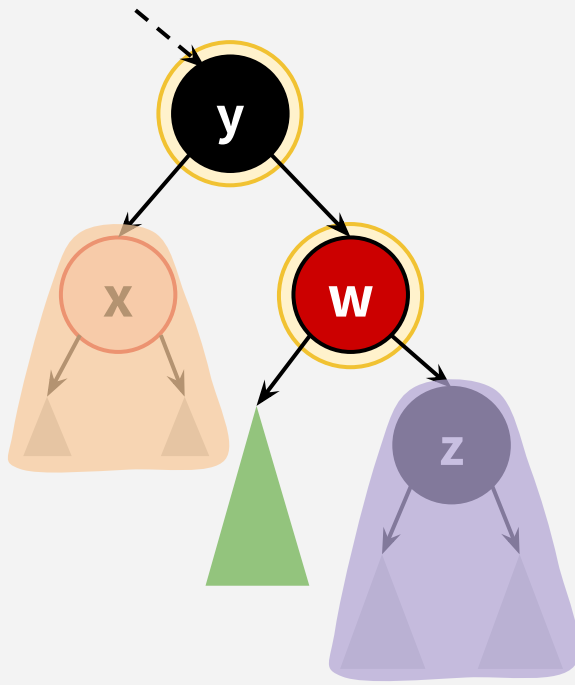
FIXING THINGS UP: CASE 2

CASE 2: parent **y** is **red**, and “uncle” **z** is **black** (or NIL)!

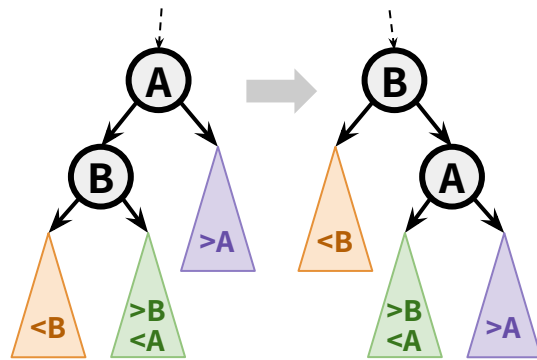
ROTATE & Recolor!

I need to rotate first, because I can't just recolor **x** or **y** to be **black**! That would mess up Property 4 - can you see why?

One rotation + **recolor**
= $O(1)$ time

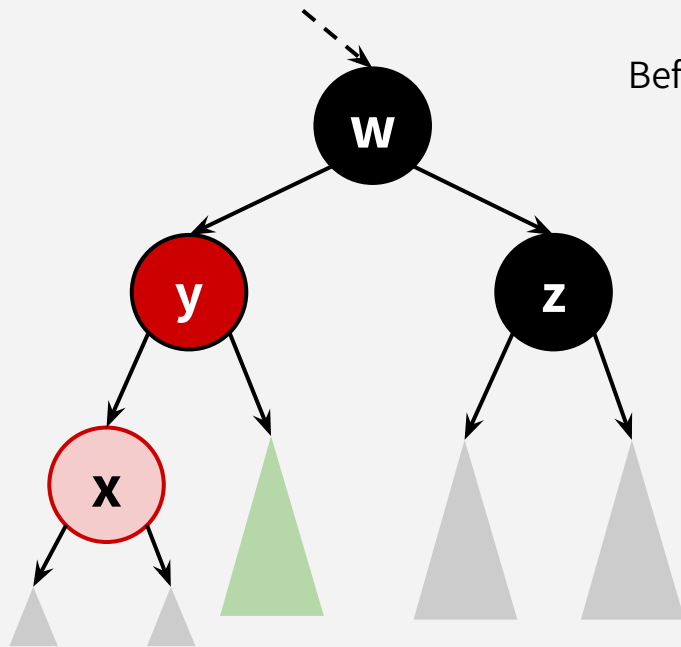


RIGHT ROTATION



FIXING THINGS UP: CASE 2

CASE 2: parent **y** is **red**, and “uncle” **z** is **black** (or NIL)!



Before rotating and recoloring

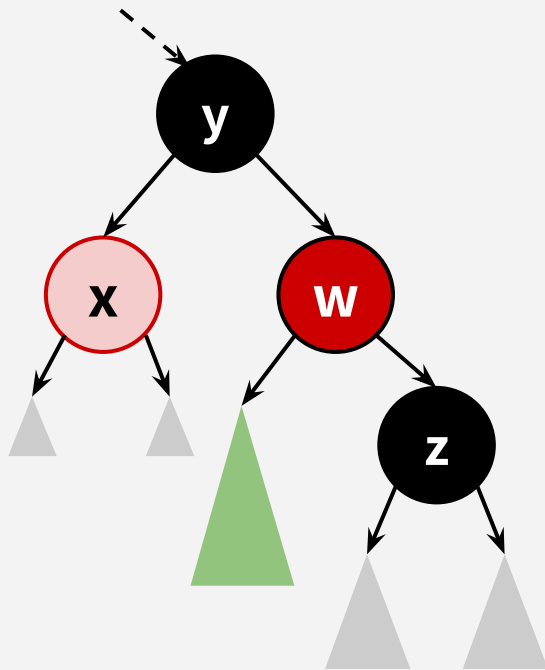
FIXING THINGS UP: CASE 2

CASE 2: parent **y** is **red**, and “uncle” **z** is **black** (or NIL)!

ROTATE & Recolor!

I need to rotate first, because I can't just recolor **x** or **y** to be **black**! That would mess up Property 4 - can you see why?

One rotation + recolor
= $O(1)$ time



After recoloring, Property 4 is maintained, and we have not propagated the “double-red” further up! We’re done!

FIXING THINGS UP: CASE 2

CASE 2: parent **y** is **red**, and “uncle” **z** is **black** (or NIL)!

Why we rotated this way should feel like magic to you right now. We needed to recolor in a way that maintained the # of black nodes on any root-NIL path, while getting rid of the “double-red”.



سوال؟

INSERT IN RB TREES

High-level plan

Insert as normal (same insert as BST), and then fix.

Fix = recolor and/or apply rotations until RB Tree properties are met.

INSERT IN RB TREES

High-level plan

Insert as normal (same insert as BST), and then fix.

Fix = recolor and/or apply rotations until RB Tree properties are met.

You are **not responsible for** the nitty-gritty details of how to insert or delete from RB Trees. And generally, I don't recommend memorizing these rotation/recolor rules.
(if you need to code up a RB Tree, just look up the pseudocode in CLRS)

INSERT IN RB TREES

High-level plan

Insert as normal (same insert as BST), and then fix.

Fix = recolor and/or apply rotations until RB Tree properties are met.

You are **not responsible for** the nitty-gritty details of how to insert or delete from RB Trees. And generally, I don't recommend memorizing these rotation/recolor rules.

(if you need to code up a RB Tree, just look up the pseudocode in CLRS)

You **should** know:

- The properties of a Red-Black tree
- Why do these properties guarantee that they are balanced?

RED-BLACK TREE HIGHLIGHTS

RED-BLACK TREES SUPPORT **SEARCH, INSERT, & DELETE**
in **$O(\log n)$** time

The key is that RB Trees always have height at most $2 \cdot \log(n+1)$.

RED-BLACK TREE HIGHLIGHTS

RED-BLACK TREES SUPPORT **SEARCH, INSERT, & DELETE**
in **$O(\log n)$** time

The key is that RB Trees always have height at most $2 \cdot \log(n+1)$.

Generally, if you need to use a BST to solve a problem, you should think of using a self-balancing BST like Red-Black Trees! Unbalanced BSTs could have worst case $O(n)$ operations.

RED-BLACK TREE HIGHLIGHTS

OPERATION	SORTED ARRAY	UNSORTED LINKED LIST	BST (WORST CASE)	BST (BALANCED)
SEARCH	$O(\log(n))$	$O(n)$	$O(n)$	$O(\log(n))$
DELETE	$O(n)$	$O(n)$	$O(n)$	$O(\log(n))$
INSERT	$O(n)$	$O(1)$	$O(n)$	$O(\log(n))$

(Balanced) Binary Search Trees can give us the best of both worlds!