

# CSE-200 Final Presentation

## Red Black Tree

---

Muhasina Ashraf - 2205002

Rifah Tasnia Islam - 2205008

Sufian Ashraf - 2205019

February 19, 2026

Department of CSE

Bangladesh University of Engineering and Technology

## We Need to Store and Search Data

- Everything is **tree-structured**

## We Need to Store and Search Data

- Everything is **tree-structured**
- **Insert** data into the structure

## We Need to Store and Search Data

- Everything is **tree-structured**
- **Insert** data into the structure
- **Delete** data efficiently

## We Need to Store and Search Data

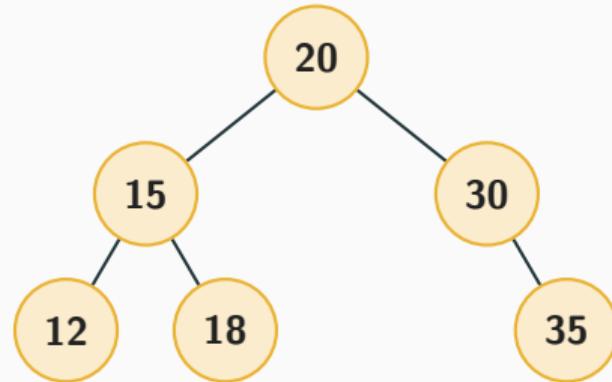
- Everything is **tree-structured**
- **Insert** data into the structure
- **Delete** data efficiently
- **Search** for data quickly

Good way to do all of this?

**Use a BST!**

## The BST Rule

How does BST decide where to put a node?



## The BST Rule

How does BST decide where to put a node?

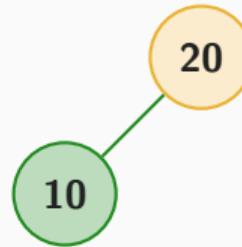


- Smaller than me? Go **Left**

## The BST Rule

How does BST decide where to put a node?

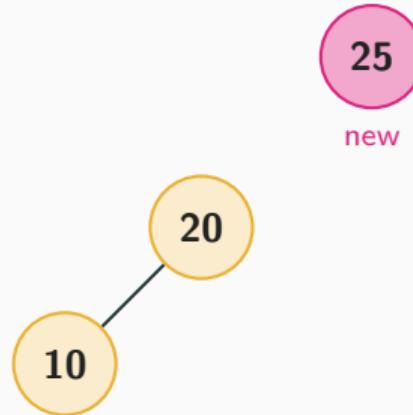
- Smaller than me? Go **Left**



## The BST Rule

How does BST decide where to put a node?

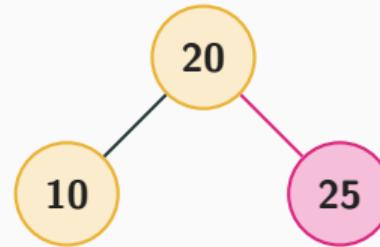
- Smaller than me? Go **Left**
- Larger than me? Go **Right**



## The BST Rule

How does BST decide where to put a node?

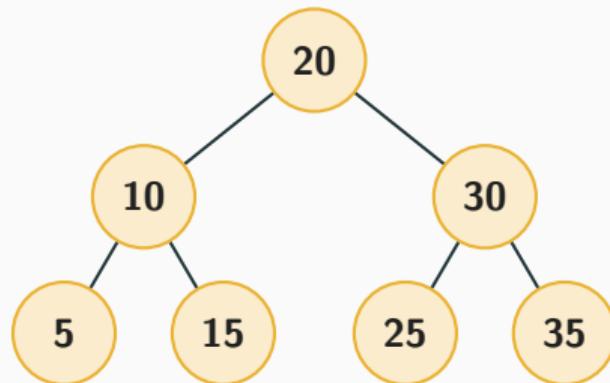
- Smaller than me? Go **Left**
- Larger than me? Go **Right**



## The BST Rule

How does BST decide where to put a node?

- Smaller than me? Go **Left**
- Larger than me? Go **Right**



**Good technique!**

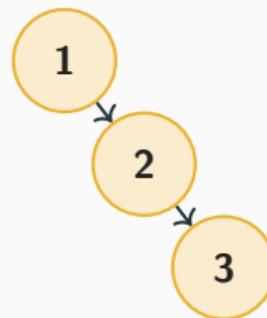
**Insert the roll numbers in a class sequentially**

1, 2, 3, 4 ...10

## Insert the roll numbers in a class sequentially

1, 2, 3, 4 ...10

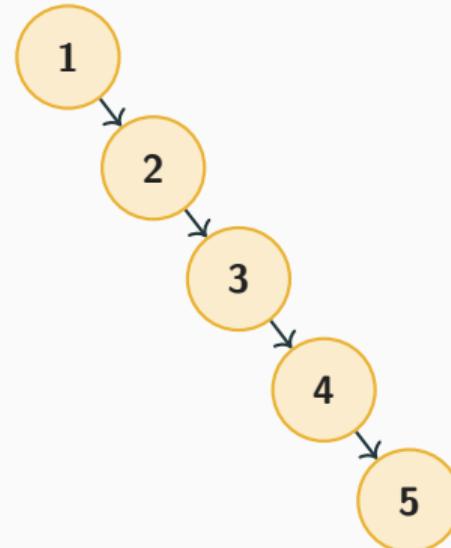
- Each goes to the **right** of the last



## Insert the roll numbers in a class sequentially

1, 2, 3, 4 ...10

- Each goes to the **right** of the last
- The tree just keeps **growing** right...

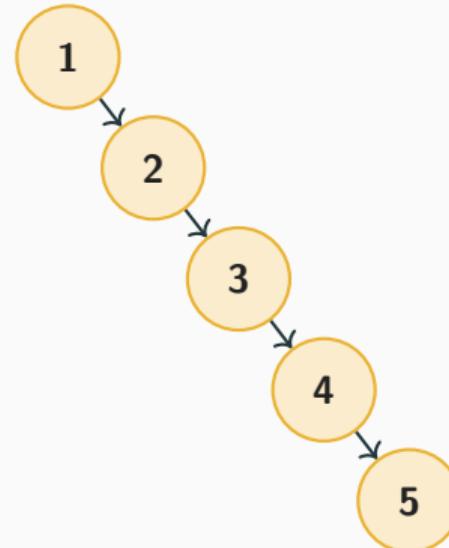


...and on

## Insert the roll numbers in a class sequentially

1, 2, 3, 4 ...10

- Each goes to the **right** of the last
- The tree just keeps **growing** right...

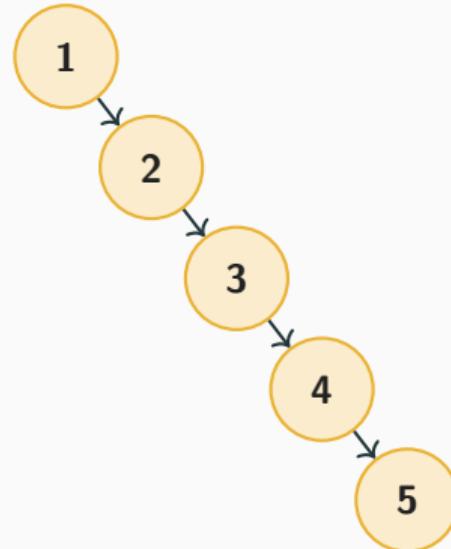


Still works!

...and on

## But, What's the Problem?

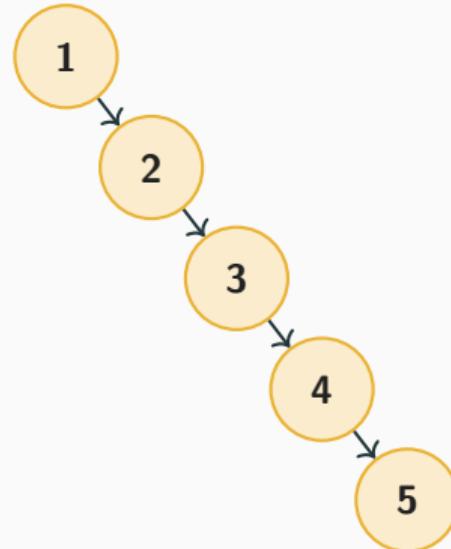
- Height becomes  $n$



...and on

## But, What's the Problem?

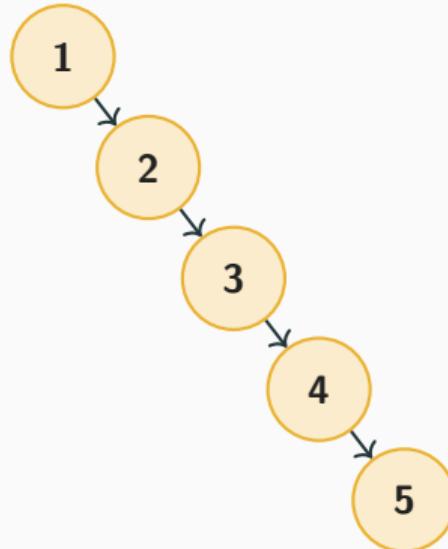
- Height becomes  $n$
- Insertion takes  $O(n)$



...and on

## But, What's the Problem?

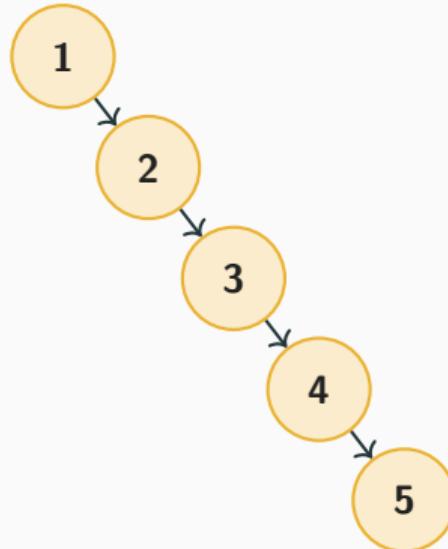
- Height becomes  $n$
- Insertion takes  $O(n)$
- Deletion takes  $O(n)$



...and on

## But, What's the Problem?

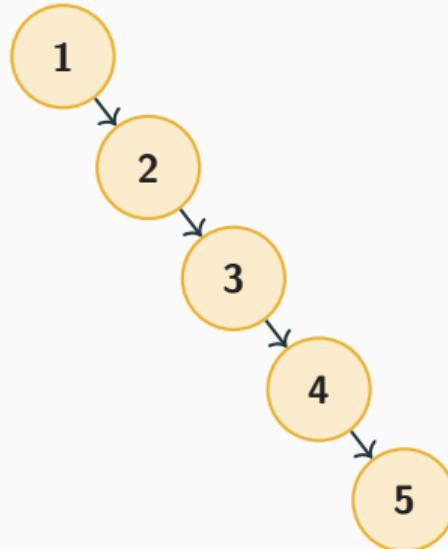
- Height becomes  $n$
- Insertion takes  $O(n)$
- Deletion takes  $O(n)$
- Search takes  $O(n)$



...and on

## But, What's the Problem?

- Height becomes  $n$
- Insertion takes  $O(n)$
- Deletion takes  $O(n)$
- Search takes  $O(n)$
- A linked list in disguise



...and on

## But, What's the Problem?

- Insertion takes  $O(n)$
- Deletion takes  $O(n)$
- Search takes  $O(n)$
- A linked list in disguise

**Time complexity becomes  $O(n)$**

## The Solution?

Use a BST that **promises** to keep its height **logarithmic**  
no matter how and what element you insert.

## The Solution?

Examples of Self-Balancing Trees:

- AVL Tree
- **Red-Black Tree**
- Splay Tree
- B-Tree

Let's look at **Red-Black** Trees



## What is Red-Black Tree

A Red-Black Tree rebalances itself by coloring nodes **red** and **black**, ensuring no two **red** nodes are **adjacent** and all **paths** have the same **black-height**, which keeps its height **logarithmic**.

## What is Red-Black Tree

A Red-Black Tree rebalances itself by coloring nodes **red** and **black**, ensuring no two **red** nodes are **adjacent** and all **paths** have the same **black-height**, which keeps its height **logarithmic**.

**Height becomes  $\log(n)$  here!**

## How does RBT do it:Properties

**Five points to remember**

## How does RBT do it:Properties

- **Property 1:** Every node is either red or black

## How does RBT do it:Properties

- **Property 1:** Every node is either red or black

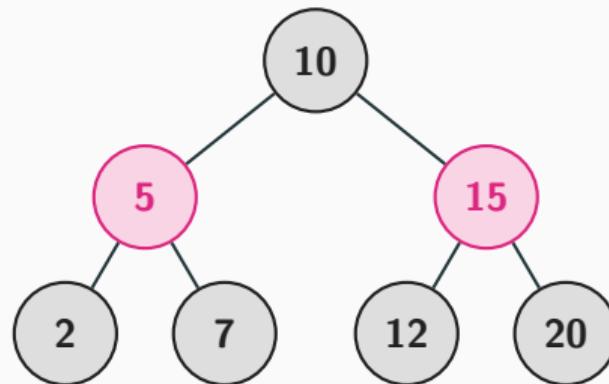
Hence, the name Red Black Tree

## How does RBT do it:Properties

- **Property 2:** Root will always be a black node

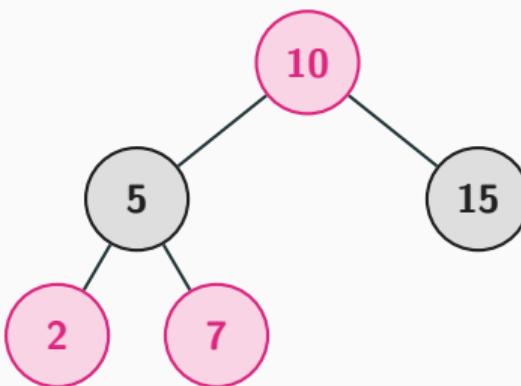
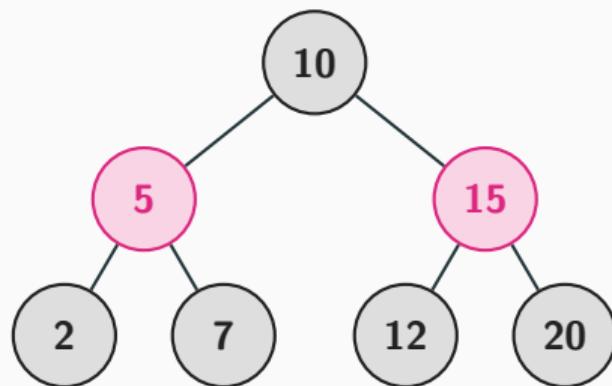
## How does RBT do it:Properties

- **Property 2:** Root will always be a black node



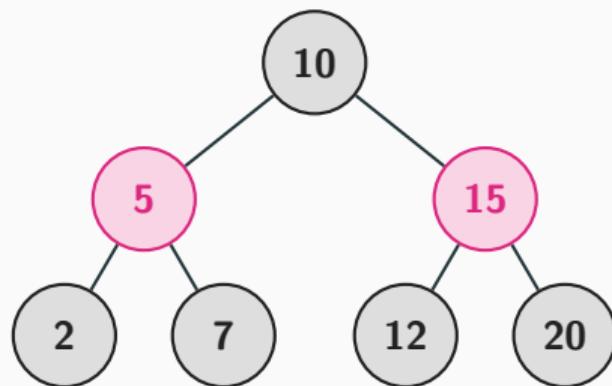
## How does RBT do it:Properties

- **Property 2:** Root will always be a black node

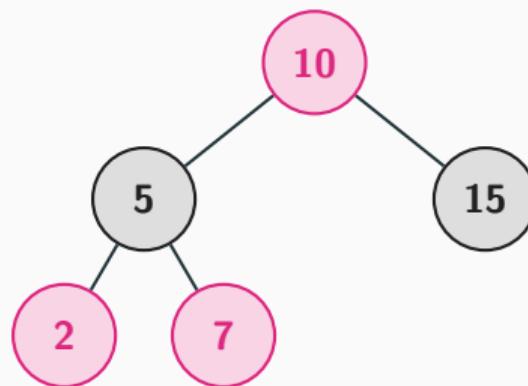


## How does RBT do it:Properties

- **Property 2:** Root will always be a black node

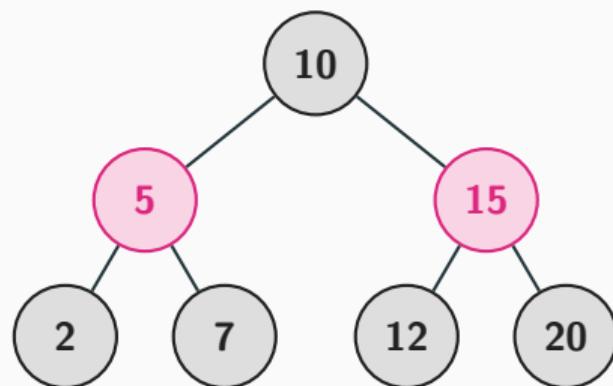


Correct

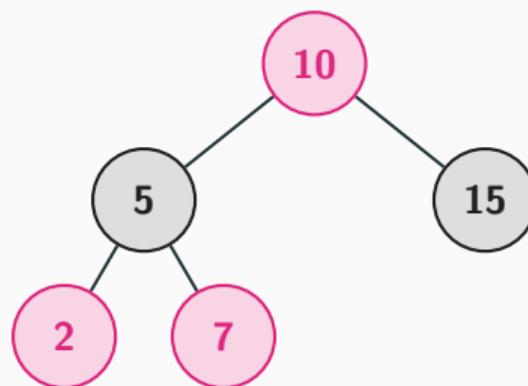


## How does RBT do it:Properties

- **Property 2:** Root will always be a black node



Correct



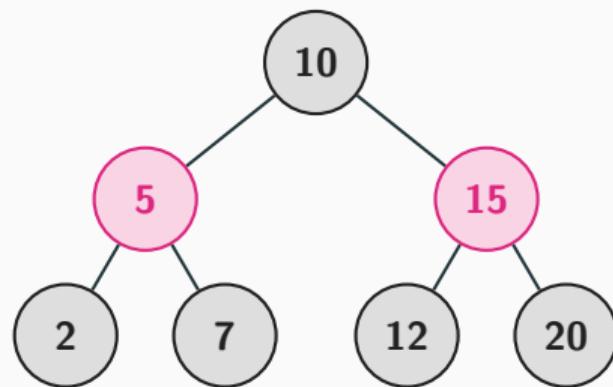
Incorrect

## How does RBT do it:Properties

- **Property 3:** Leaves will either be black or NIL

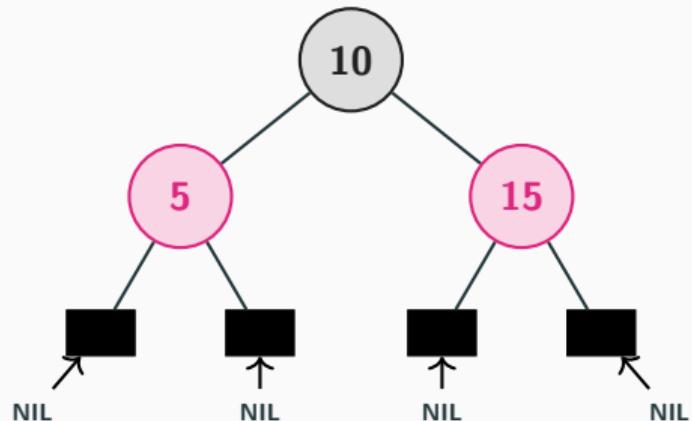
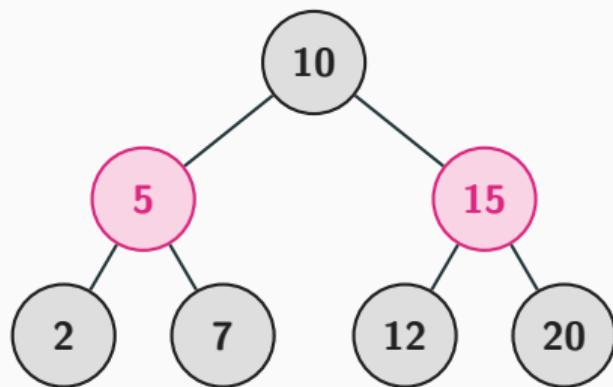
## How does RBT do it:Properties

- **Property 3:** Leaves will either be black or NIL



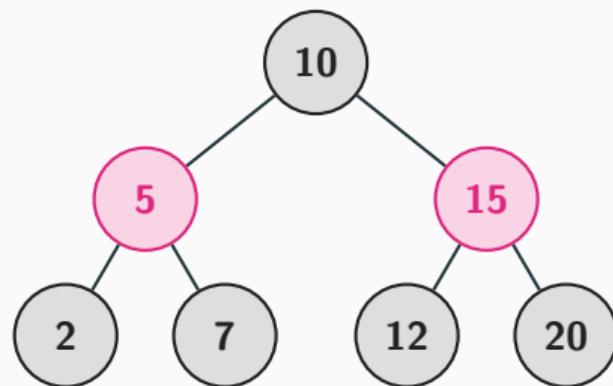
## How does RBT do it:Properties

- **Property 3:** Leaves will either be black or NIL

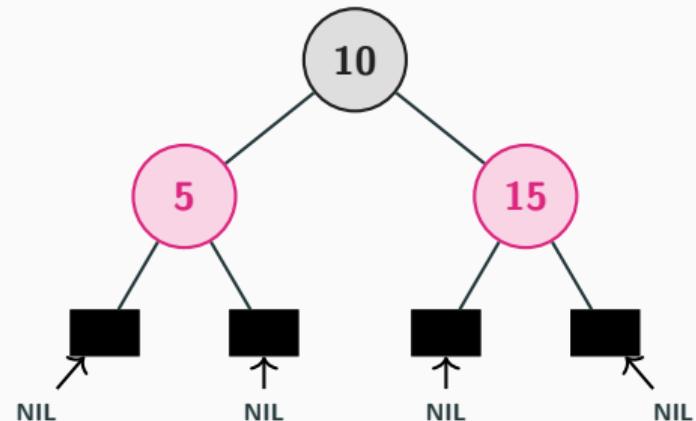


## How does RBT do it:Properties

- **Property 3:** Leaves will either be black or NIL

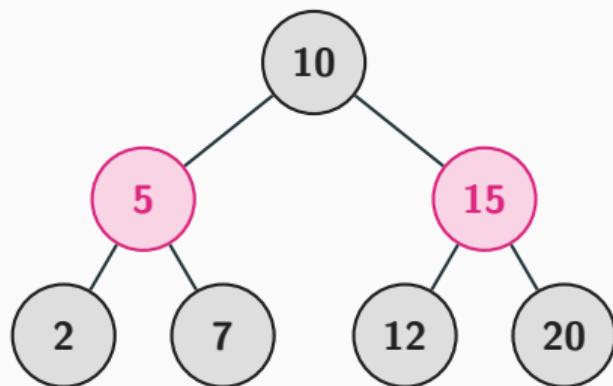


Black Leaves

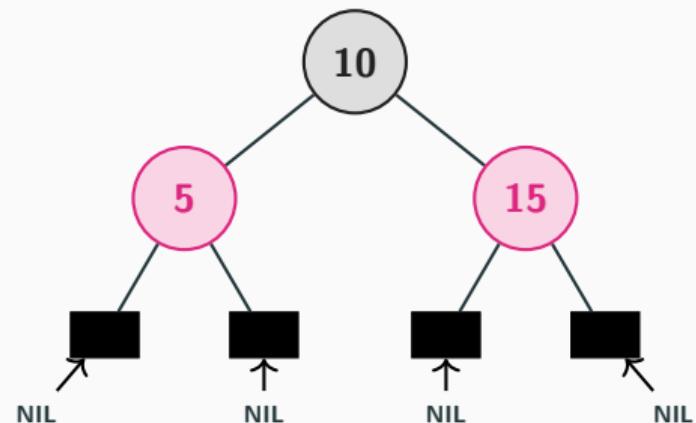


## How does RBT do it:Properties

- **Property 3:** Leaves will either be black or NIL



Black Leaves



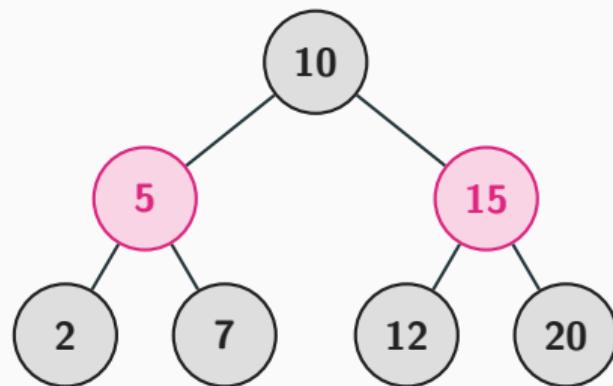
NIL nodes (counted as Black)

## How does RBT do it:Properties

- **Property 4:** There will be no two consecutive red nodes

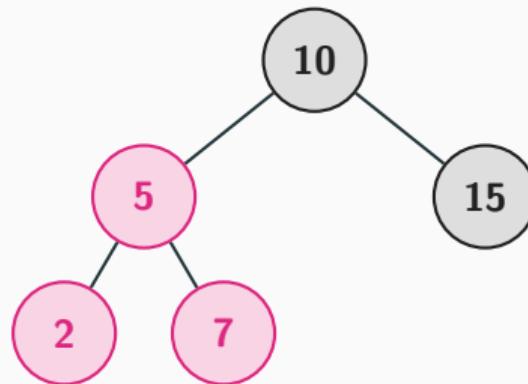
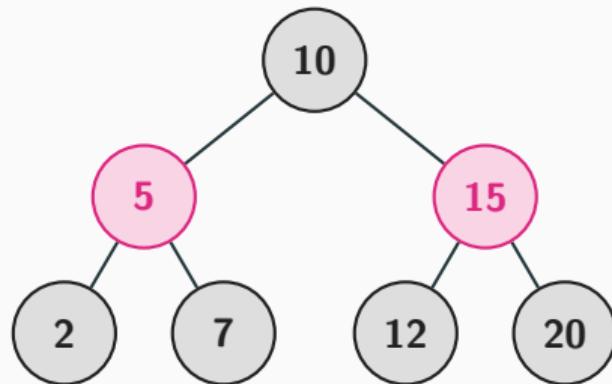
## How does RBT do it:Properties

- **Property 4:** There will be no two consecutive red nodes



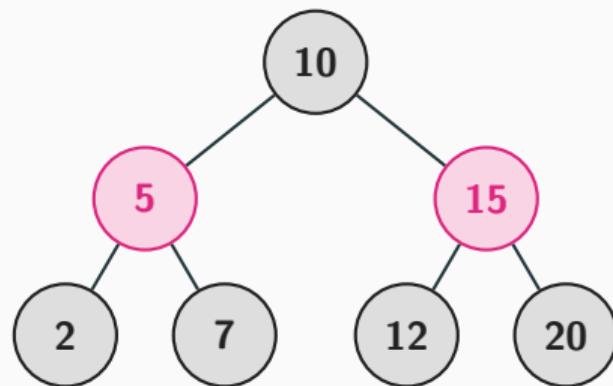
## How does RBT do it:Properties

- **Property 4:** There will be no two consecutive red nodes

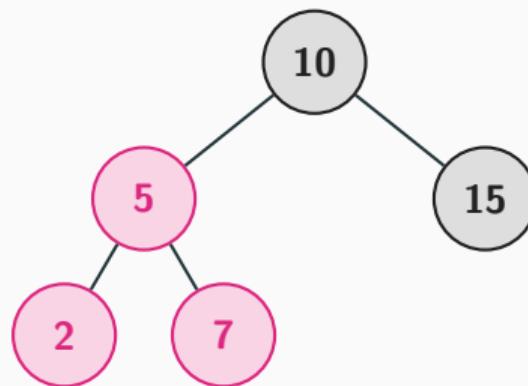


## How does RBT do it:Properties

- **Property 4:** There will be no two consecutive red nodes

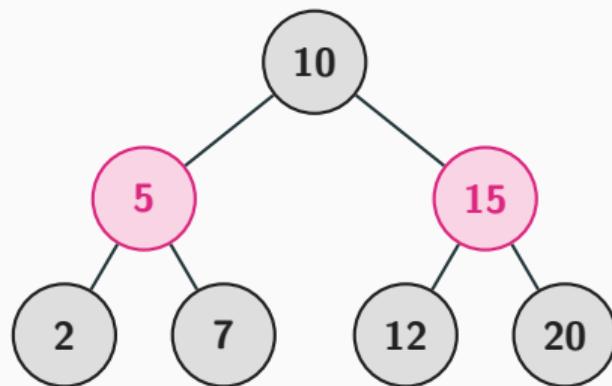


Correct

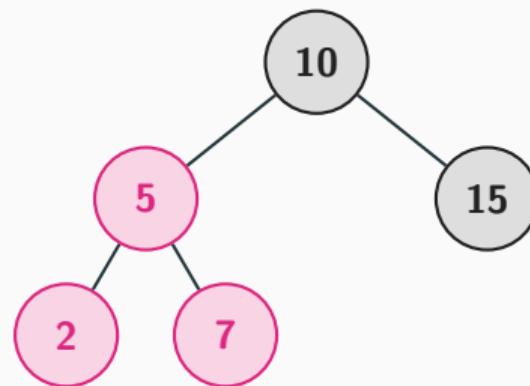


## How does RBT do it:Properties

- **Property 4:** There will be no two consecutive red nodes



Correct



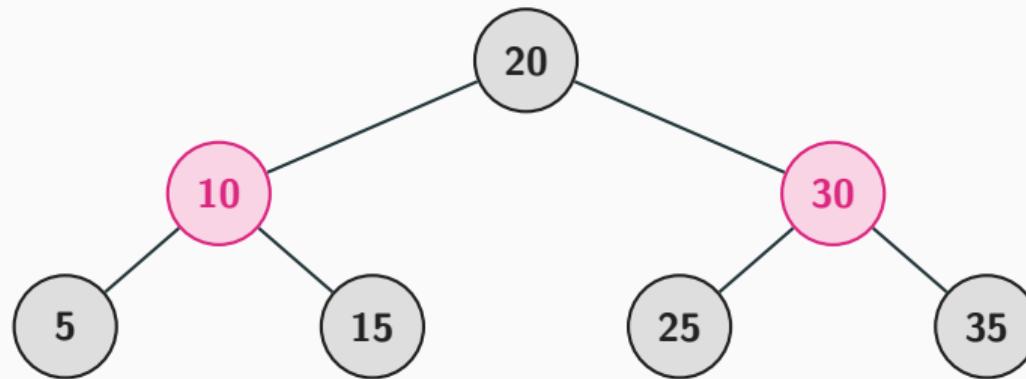
Incorrect

## How does RBT do it:Properties

- **Property 5:** From a given node, the number of black nodes in any given path will always be same for that node

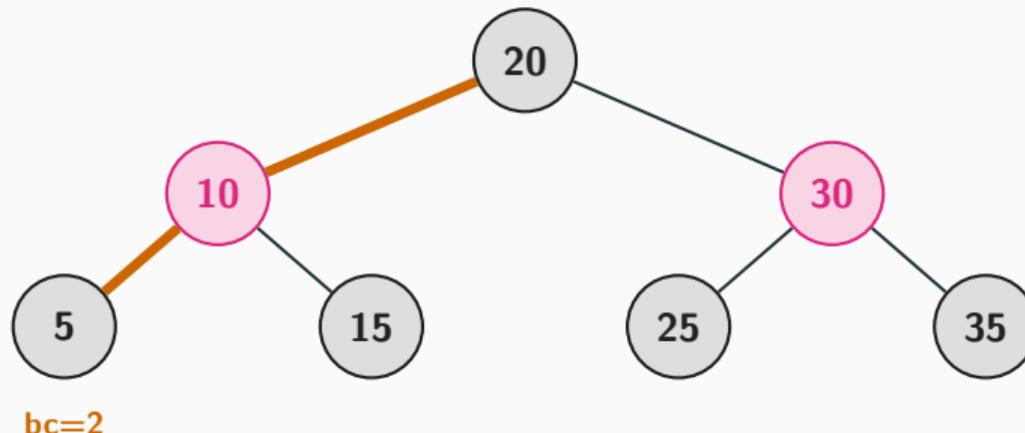
## How does RBT do it:Properties

- **Property 5:** From a given node, the number of black nodes in any given path will always be same for that node



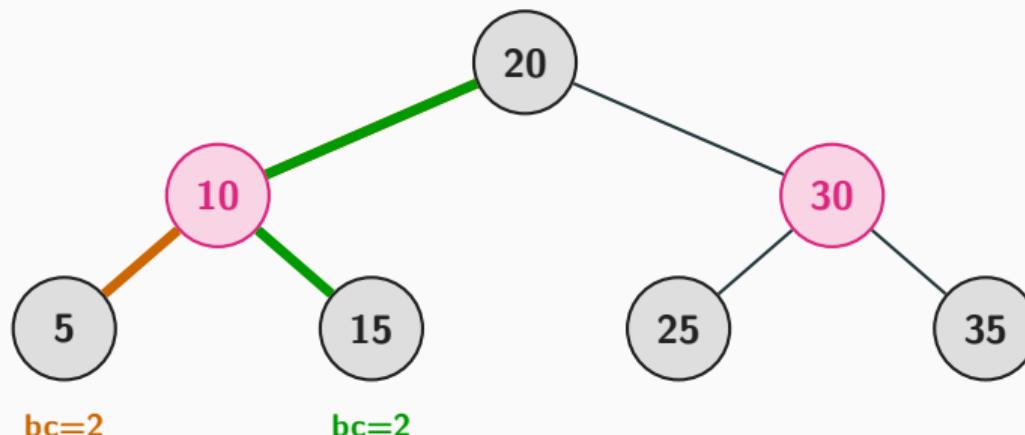
## How does RBT do it:Properties

- **Property 5:** From a given node, the number of black nodes in any given path will always be same for that node



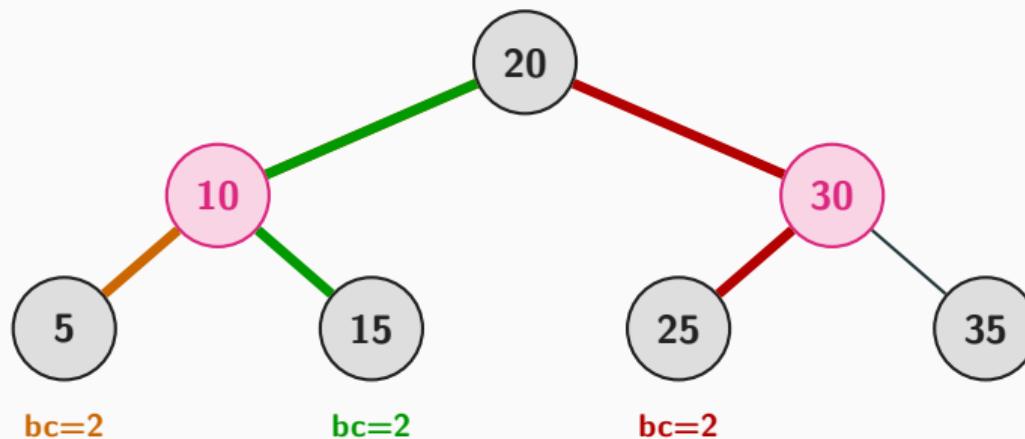
## How does RBT do it:Properties

- **Property 5:** From a given node, the number of black nodes in any given path will always be same for that node



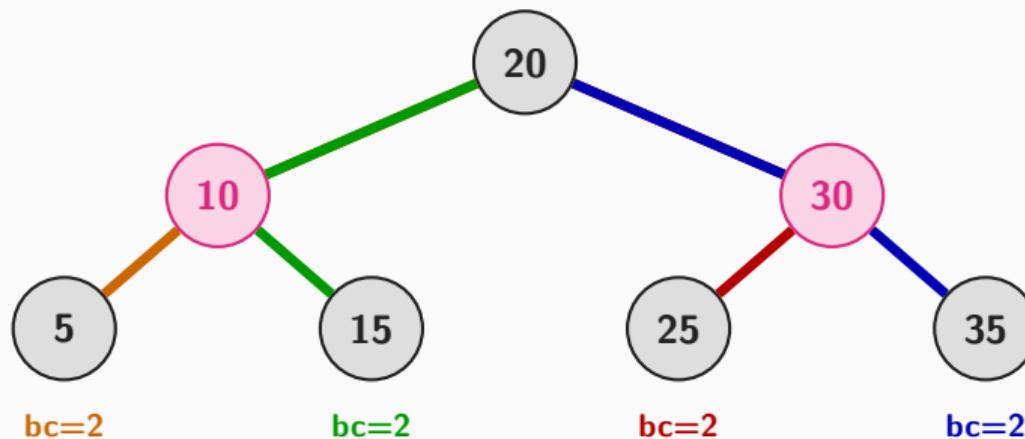
## How does RBT do it:Properties

- **Property 5:** From a given node, the number of black nodes in any given path will always be same for that node



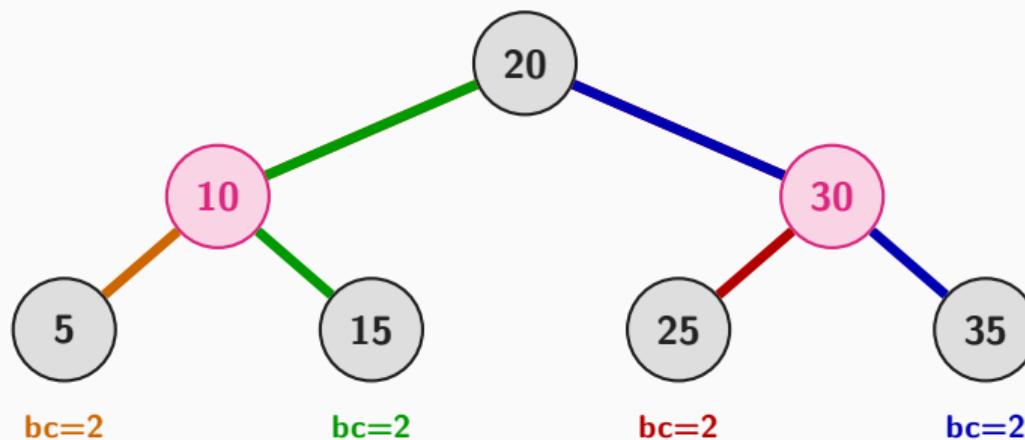
## How does RBT do it:Properties

- **Property 5:** From a given node, the number of black nodes in any given path will always be same for that node



## How does RBT do it:Properties

- **Property 5:** From a given node, the number of black nodes in any given path will always be same for that node



All paths from root have same black count = 2

## RBT Operations

Now, How do these points ensure the "rebalancing" feature of Red Black Tree?

Now, How do these points ensure the "rebalancing" feature of Red Black Tree?

**Let's see some operations....**

## Insertion

Insert node x in a Red Black Tree

## Insertion

Insert node x in a Red Black Tree

### Pseudocode

```
color[x] = RED  
  
y = root[T]  
  
while y ≠ NIL do  
    if key[x] > key[y]  
        y = right[y]  
    else  
        y = left[y]
```

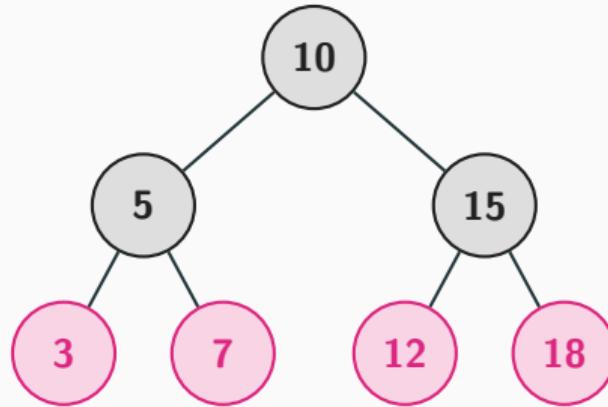
## Insertion

Insert node x in a Red Black Tree

### Pseudocode

```
color[x] = RED  
  
y = root[T]  
  
while y ≠ NIL do  
    if key[x] > key[y]  
        y = right[y]  
    else  
        y = left[y]
```

Existing RBT

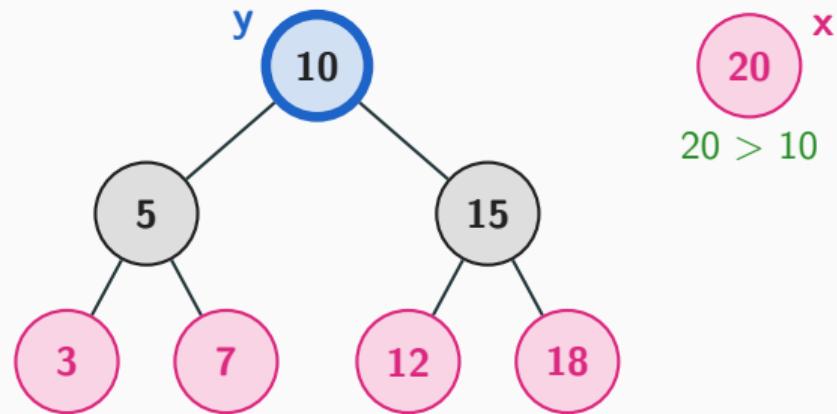


# Insertion

Insert node x in a Red Black Tree

## Pseudocode

```
color[x] = RED  
  
y = root[T]  
  
while y ≠ NIL do  
    if key[x] > key[y]  
        y = right[y]  
    else  
        y = left[y]
```

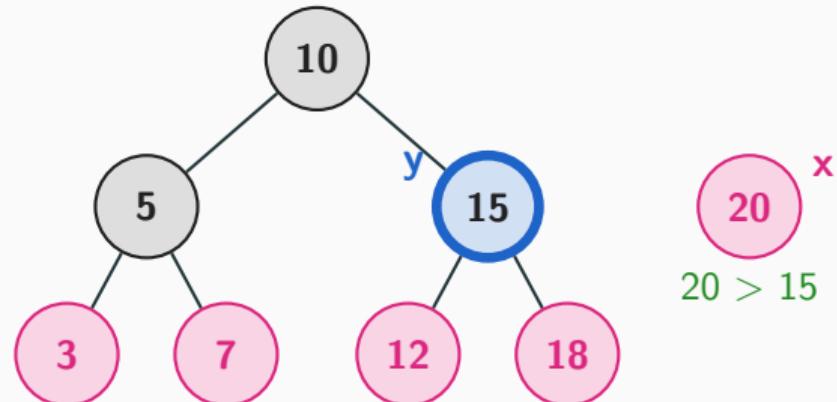


## Insertion

Insert node x in a Red Black Tree

### Pseudocode

```
color[x] = RED  
  
y = root[T]  
  
while y ≠ NIL do  
    if key[x] > key[y]  
        y = right[y]  
    else  
        y = left[y]
```

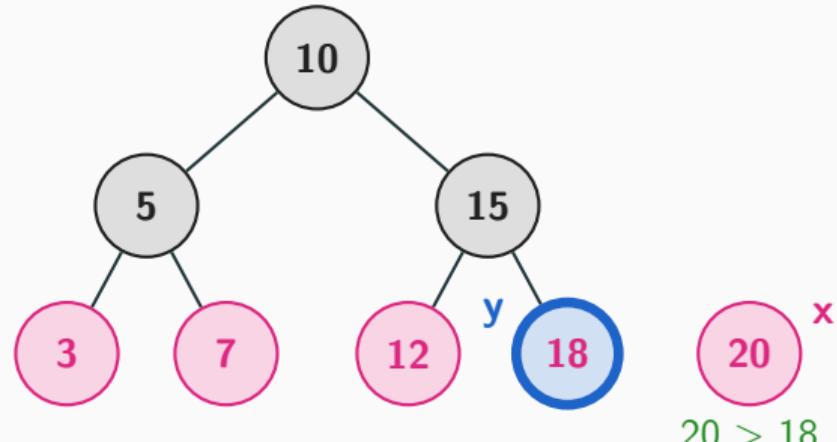


## Insertion

Insert node x in a Red Black Tree

### Pseudocode

```
color[x] = RED  
  
y = root[T]  
  
while y ≠ NIL do  
    if key[x] > key[y]  
        y = right[y]  
    else  
        y = left[y]
```

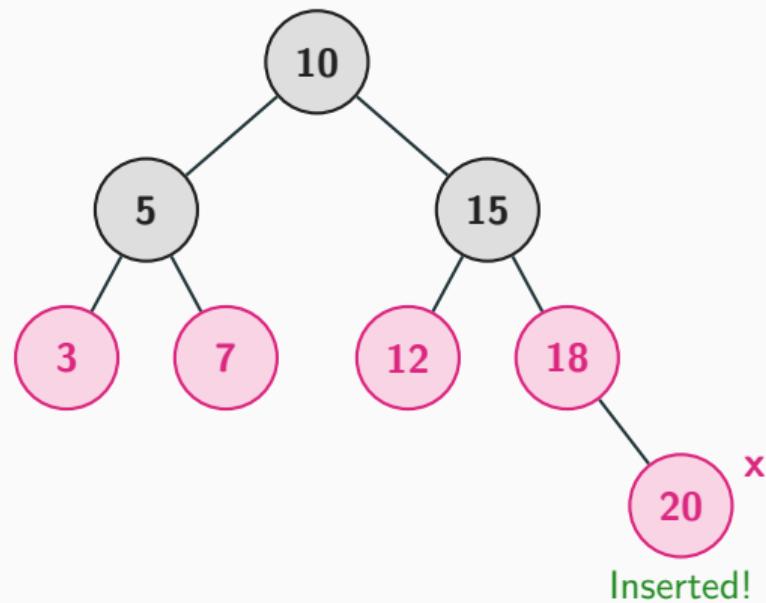


## Insertion

Insert node x in a Red Black Tree

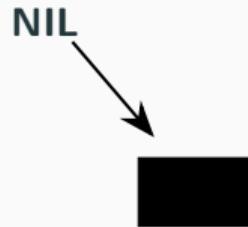
### Pseudocode

```
color[x] = RED  
  
y = root[T]  
  
while y ≠ NIL do  
    if key[x] > key[y]  
        y = right[y]  
    else  
        y = left[y]
```



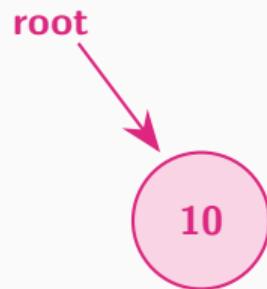
## What can go wrong

Insert 10



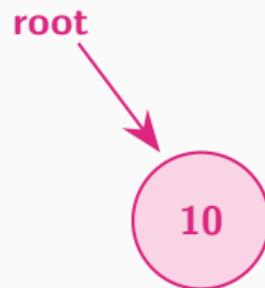
## What can go wrong

Insert 10



## What can go wrong

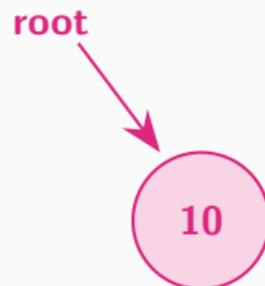
Insert 10



Root can't be RED

## What can go wrong

Insert 10

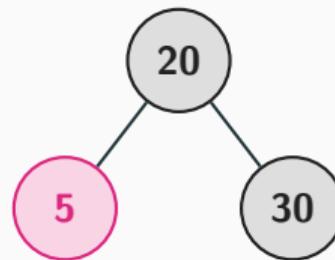
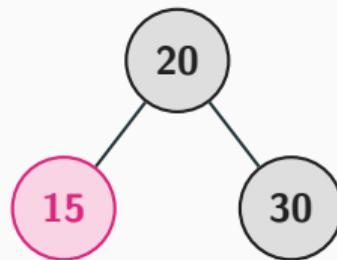
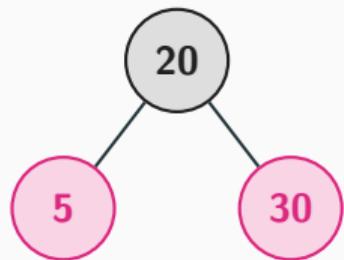


Root can't be RED

Case 1

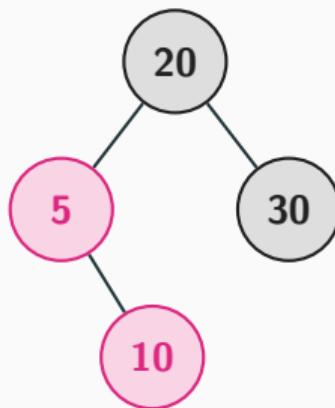
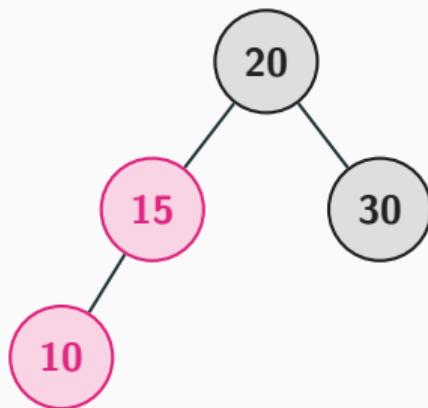
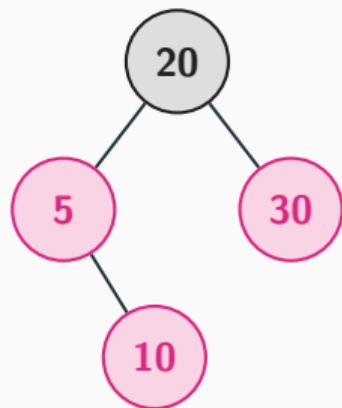
## What can go wrong

Insert 10



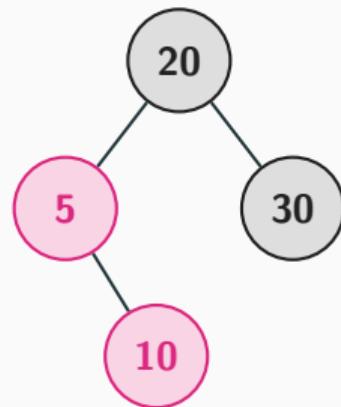
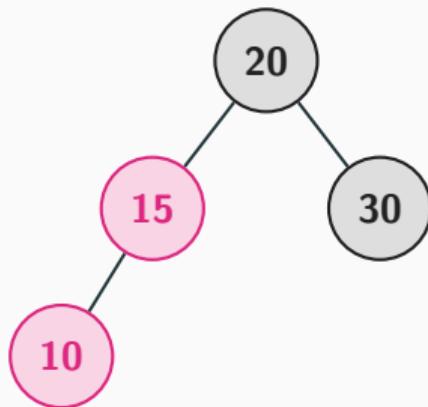
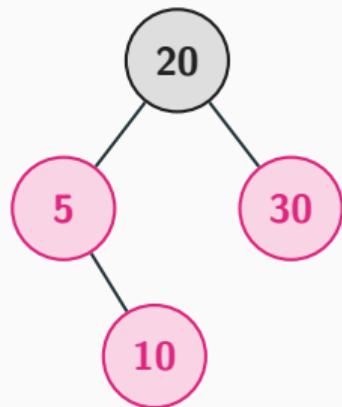
## What can go wrong

Insert 10



## What can go wrong

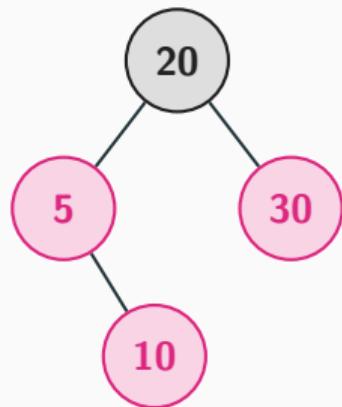
Insert 10



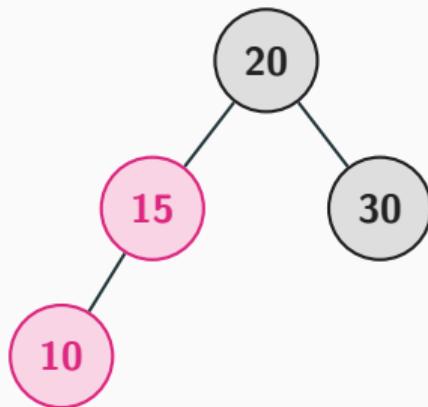
Case 2

## What can go wrong

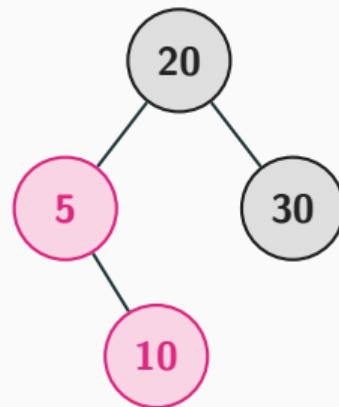
Insert 10



Case 2

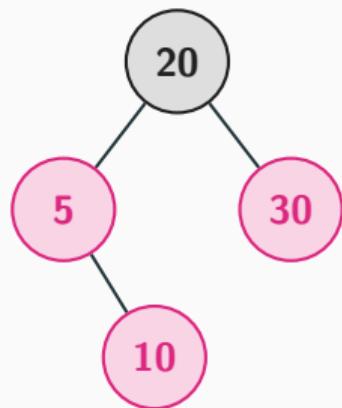


Case 3

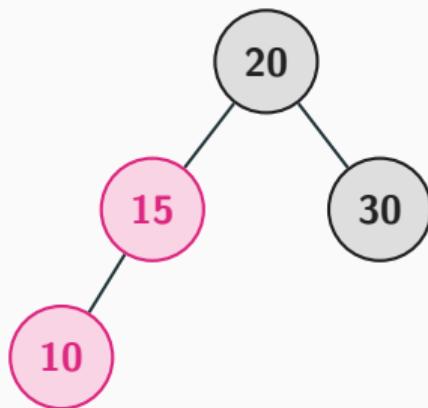


## What can go wrong

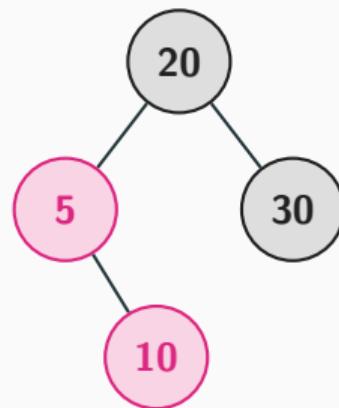
Insert 10



Case 2



Case 3

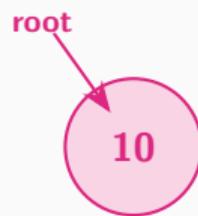


Case 4

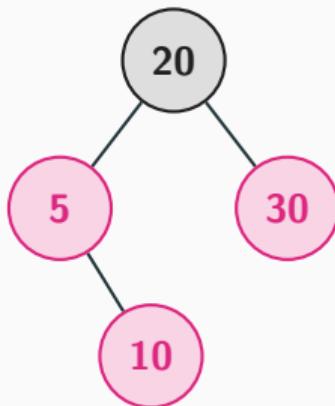
## Insertion Violation Cases

Four main violation cases in Red-Black Tree insertion:

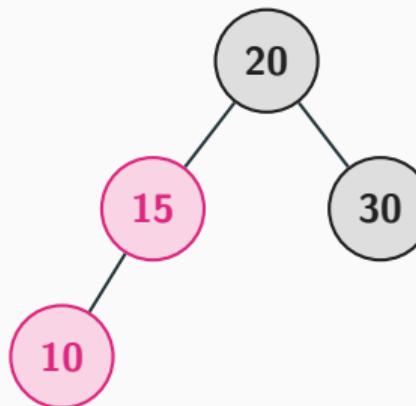
Case 1



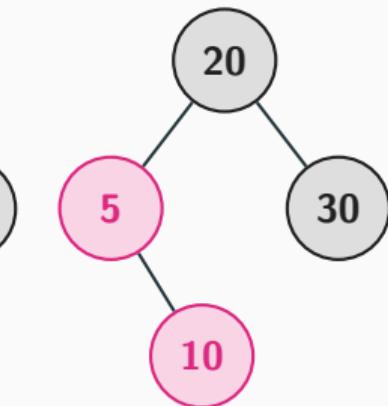
Case 2



Case 3



Case 4



How to solve this?

## Solutions to Violations

Case 1

 **Violation**

Root is **RED** — Property 2 broken

 **Fix**

Recolor root to **BLACK**

## Solutions to Violations

Case 1

### Violation

Root is **RED** — Property 2 broken

### Fix

Recolor root to **BLACK**

Case 2

### Violation

Uncle is **RED** — two reds adjacent

### Fix

Recolor parent, uncle **BLACK**;  
grandparent **RED**

## Solutions to Violations

Case 1

### Violation

Root is **RED** — Property 2 broken

### Fix

Recolor root to **BLACK**

Case 2

### Violation

Uncle is **RED** — two reds adjacent

### Fix

Recolor parent, uncle **BLACK**;  
grandparent **RED**

Case 3

### Violation

Uncle is **BLACK** — Right-Right

### Fix

Left rotate at grandparent, then  
recolor

## Solutions to Violations

Case 1

### ⚠️ Violation

Root is **RED** — Property 2 broken

### 🔧 Fix

Recolor root to **BLACK**

Case 2

### ⚠️ Violation

Uncle is **RED** — two reds adjacent

### 🔧 Fix

Recolor parent, uncle **BLACK**;  
grandparent **RED**

Case 3

### ⚠️ Violation

Uncle is **BLACK** — Right-Right

### 🔧 Fix

Left rotate at grandparent, then  
recolor

Case 4

### ⚠️ Violation

Uncle is **BLACK** — Left-Right

### 🔧 Fix

Left rotate at parent, then apply Case  
3

## Remember This Problem?

Let's try inserting **1, 2, 3, 4, 5** again...

But this time in a **Red-Black Tree!**

## Remember This Problem?

Let's try inserting **1, 2, 3, 4, 5** again...

But this time in a **Red-Black Tree!**

**Watch the magic happen!**

## Insert 1

- First node is always **root**
- Insert as **RED** (default color)

## Insert 1

- First node is always **root**
- Insert as **RED** (default color)

After Insert



## Insert 1

- First node is always **root**
- Insert as **RED** (default color)
- But Root cannot be **RED!**

**Property 2 violated - Case : 1**

**Violation — Root is RED!**



## Insert 1

- First node is always **root**
  - Insert as **RED** (default color)
  - But Root cannot be **RED!**
- Property 2 violated - Case : 1**
- Recolor root to **BLACK**
  - **Fixed!**

After Recolor



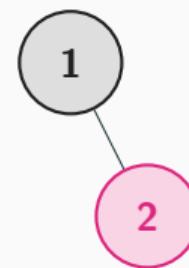
## Insert 2

- Right child of 1
- Insert as **RED** (default color)

## Insert 2

- Right child of 1
- Insert as **RED** (default color)
- Parent is BLACK — **no violation ✓**

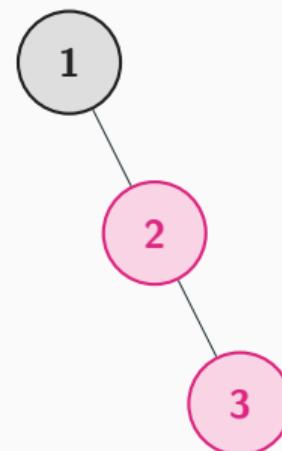
After Insert



## Insert 3

- Right child of 2
- Insert as **RED** (default)

After Insert

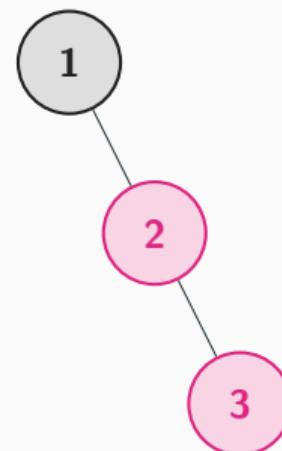


## Insert 3

- Right child of 2
- Insert as **RED** (default)
- Uncle is **NIL/BLACK**
- **Case: 3**

Left rotate at node 1

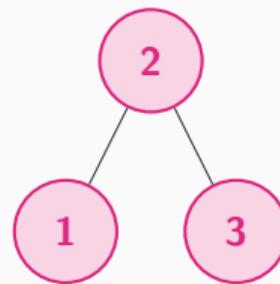
**Violation Found — Two RED in a row!**



## Insert 3

- Right child of 2
- Insert as **RED** (default)
- Uncle is **NIL/BLACK**
- **Case: 3**  
Left rotate at node 1
- Recolor: 2 → **BLACK**, children  
→ **RED**

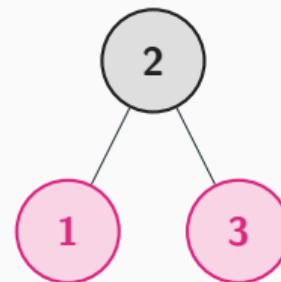
After Left Rotation



## Insert 3

- Right child of 2
- Insert as **RED** (default)
- Uncle is **NIL/BLACK**
- **Case: 3**
  - Left rotate at node 1
- Recolor: 2 → **BLACK**, children → **RED**
- **Fixed!**

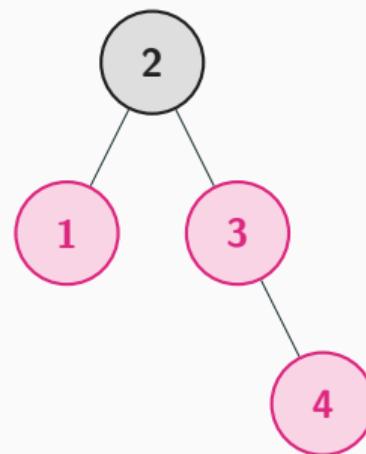
After Recolor



## Insert 4

- Right child of 3
- Insert as **RED** (default)

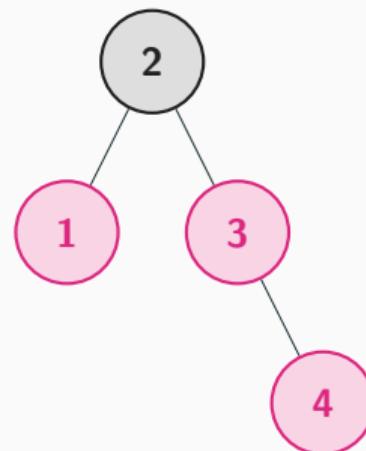
After Insert



## Insert 4

- Right child of 3
- Insert as **RED** (default)
- Uncle (node 1) is **RED**
- **CASE 2**  
Uncle is RED — just recolor!

**Violation Found — Uncle is RED**



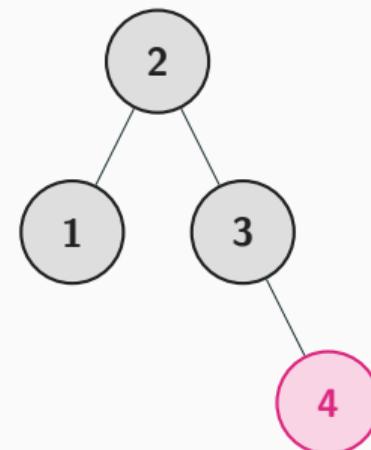
## Insert 4

- Right child of 3
- Insert as **RED** (default)
- Uncle (node 1) is **RED**
- **CASE 2**
  - Uncle is RED — just recolor!
- Recolor: parent & uncle →  
**BLACK**
- Grandparent stays **BLACK**

## Insert 4

- Right child of 3
- Insert as **RED** (default)
- Uncle (node 1) is **RED**
- **CASE 2**
  - Uncle is RED — just recolor!
- Recolor: parent & uncle → **BLACK**
- Grandparent stays **BLACK**
- **Fixed!**

After Recolor



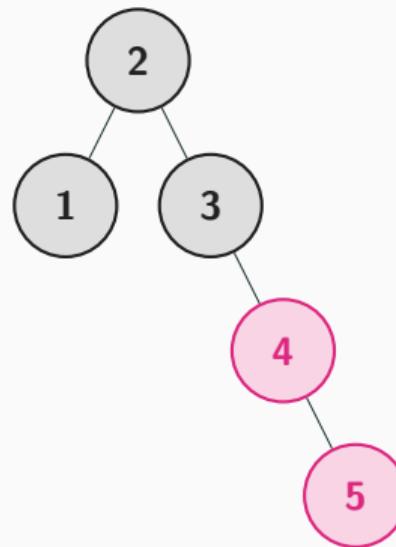
## Insert 5

- Right child of 4
- Insert as **RED** (default)

## Insert 5

- Right child of 4
- Insert as **RED** (default)

After Insert

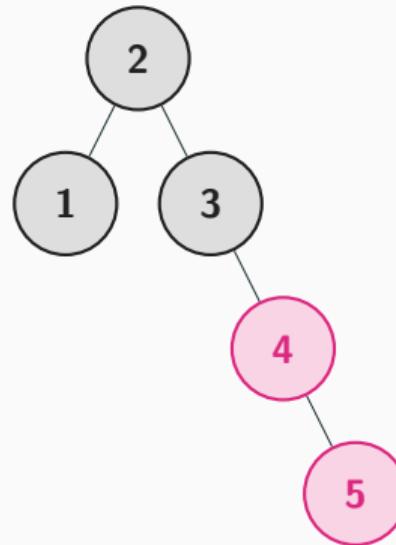


## Insert 5

- Right child of 4
- Insert as **RED** (default)
- Uncle (node 1) is **BLACK**
- **CASE 3**

Uncle is **BLACK** — rotate & recolor!

**Violation — Two RED in a row!**



## Insert 5

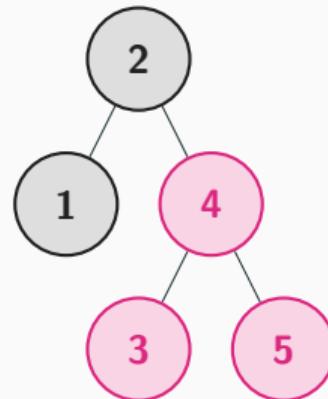
- Right child of 4
- Insert as **RED** (default)
- Uncle (node 1) is **BLACK**

### CASE 3

Uncle is **BLACK** — rotate & recolor!

- Left rotate at node 3 → 4 moves up

After Left Rotation



## Insert 5

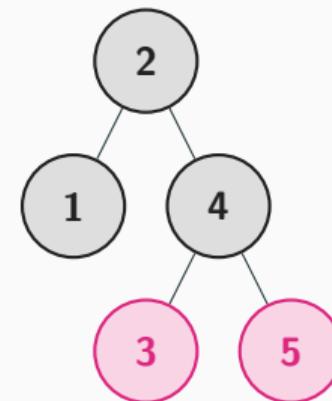
- Right child of 4
- Insert as **RED** (default)
- Uncle (node 1) is **BLACK**

### CASE 3

Uncle is **BLACK** — rotate & recolor!

- Left rotate at node 3 → 4 moves up
- Recolor: 4 → **BLACK**, children → **RED**
- **We're done!**

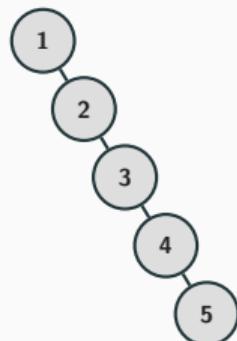
After Recolor



## BST vs. Red-Black Tree

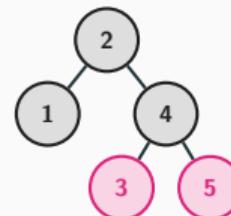
Inserting  $\{1, 2, 3, 4, 5\}$  in order

Regular BST



Height = 5 •  $O(n)$

Red-Black Tree



Height = 3 •  $O(\log n)$

**Insertion is the easy half**

**Now, What happens when we delete a node?**

**Deletion is even more... interesting!**

**Deletion is even more... interesting!**

Deleting a **RED** node

Deleting a **BLACK** node

# Deletion is even more... interesting!

Deleting a **RED** node

Deleting a **BLACK** node

- No problem!

# Deletion is even more... interesting!

### Deleting a **RED** node

- No problem!
- Just remove it

### Deleting a **BLACK** node

# Deletion is even more... interesting!

### Deleting a **RED** node

- No problem!
- Just remove it
- **Properties still hold**

### Deleting a **BLACK** node

# Deletion is even more... interesting!

### Deleting a **RED** node

- No problem!
- Just remove it
- **Properties still hold**

### Deleting a **BLACK** node

- Oh boy...

# Deletion is even more... interesting!

### Deleting a **RED** node

- No problem!
- Just remove it
- **Properties still hold**

### Deleting a **BLACK** node

- Oh boy...
- Black height changes!

# Deletion is even more... interesting!

### Deleting a **RED** node

- No problem!
- Just remove it
- **Properties still hold**

### Deleting a **BLACK** node

- Oh boy...
- Black height changes!
- **Need “double black” fix**

# Deletion is even more... interesting!

### Deleting a **RED** node

- No problem!
- Just remove it
- **Properties still hold**

### Deleting a **BLACK** node

- Oh boy...
- Black height changes!
- **Need “double black” fix**
- Complex cases

# Deletion is even more... interesting!

### Deleting a **RED** node

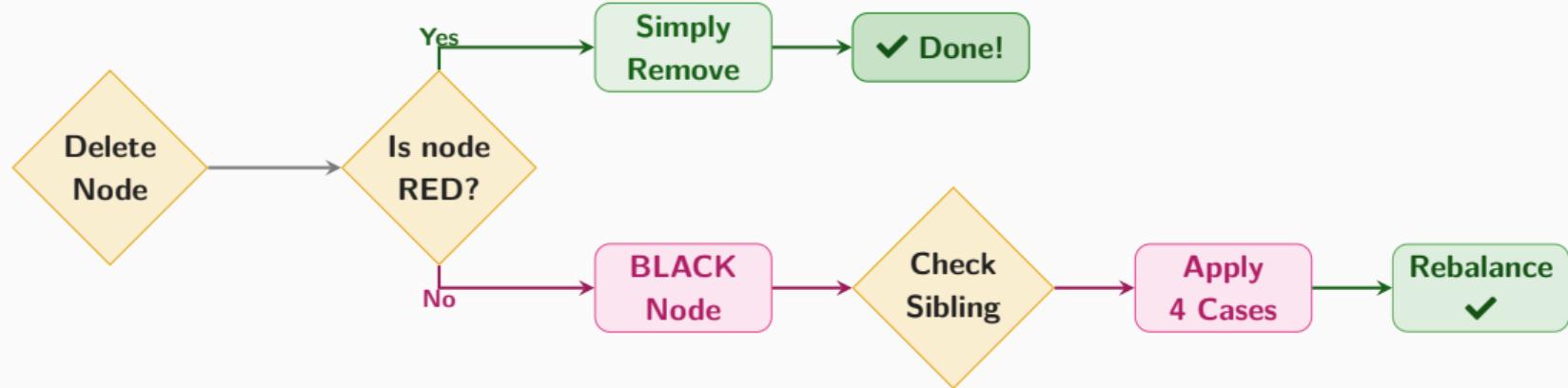
- No problem!
- Just remove it
- **Properties still hold**

### Deleting a **BLACK** node

- Oh boy...
- Black height changes!
- **Need “double black” fix**
- Complex cases

Let's see both cases...

## Deletion Decision Flowchart

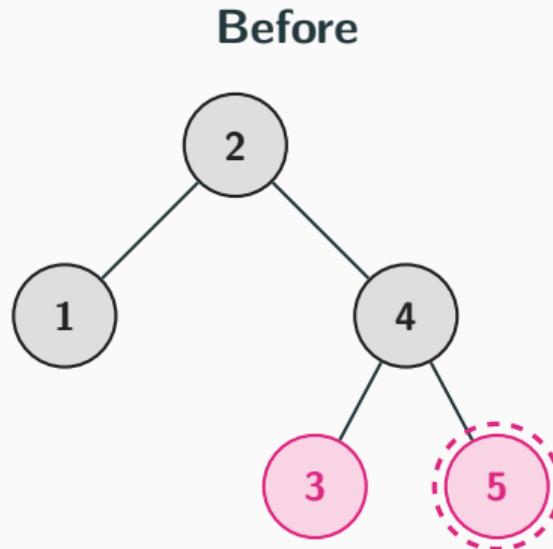


**Top path (RED node)** = straightforward

**Bottom path (BLACK node)** = complex

## Case 1: Deleting a RED Node

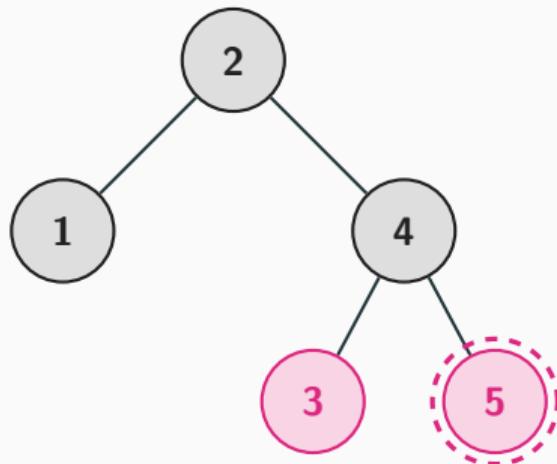
Delete node **5** from the tree



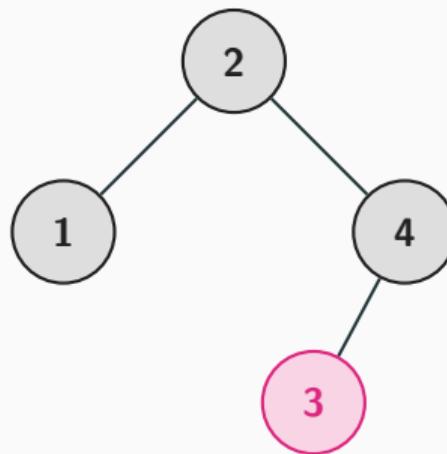
## Case 1: Deleting a RED Node

Delete node **5** from the tree

Before



After



## Case 1: Deleting a **RED** Node

Is it done?

## Case 1: Deleting a **RED** Node

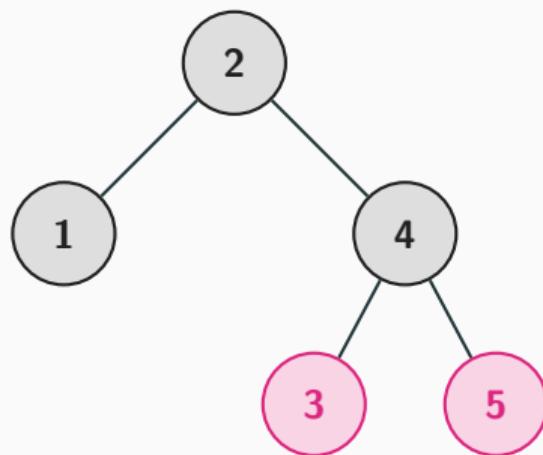
**Is it done?**

Let's check the black height.

## Case 1: Black-Height Stays the Same

Every path still has  $\text{bc} = 2$  black nodes after removing 5

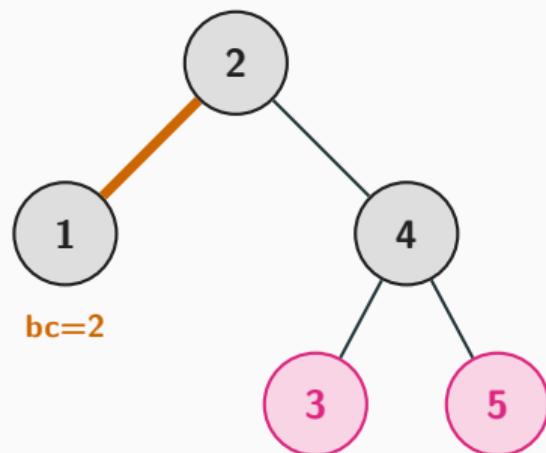
Before (with node 5)



## Case 1: Black-Height Stays the Same

Every path still has  $\text{bc} = 2$  black nodes after removing 5

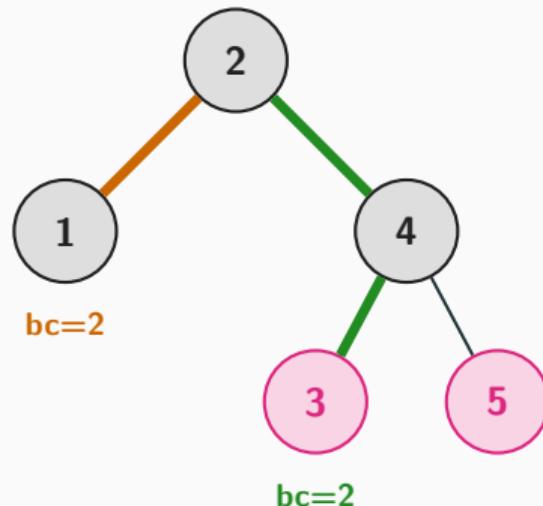
Before (with node 5)



## Case 1: Black-Height Stays the Same

Every path still has  $\text{bc} = 2$  black nodes after removing 5

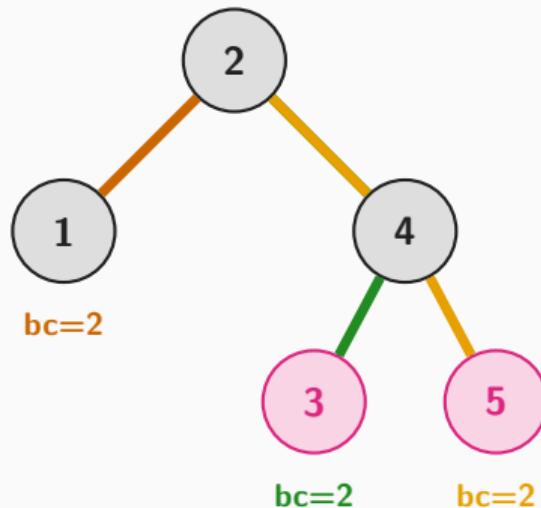
Before (with node 5)



## Case 1: Black-Height Stays the Same

Every path still has  $\text{bc} = 2$  black nodes after removing 5

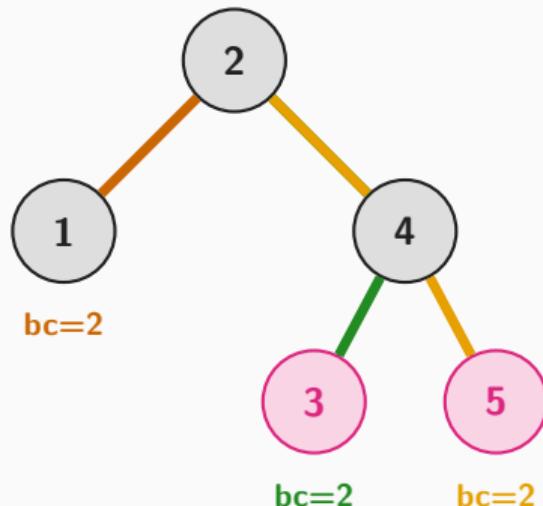
Before (with node 5)



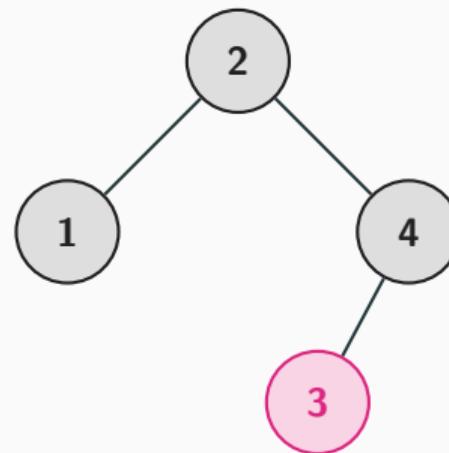
## Case 1: Black-Height Stays the Same

Every path still has  $\text{bc} = 2$  black nodes after removing 5

Before (with node 5)



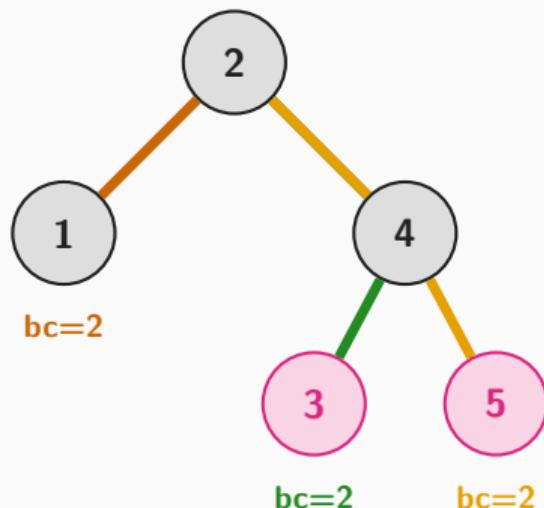
After (node 5 removed)



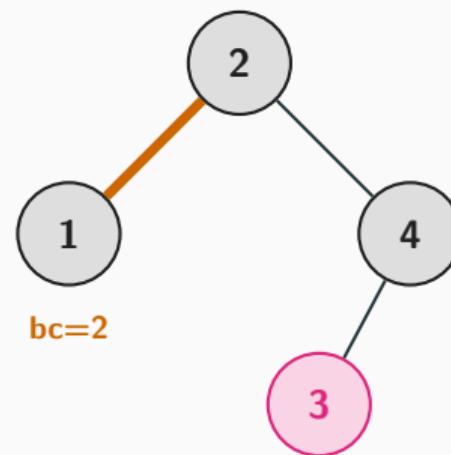
## Case 1: Black-Height Stays the Same

Every path still has  $\text{bc} = 2$  black nodes after removing 5

Before (with node 5)



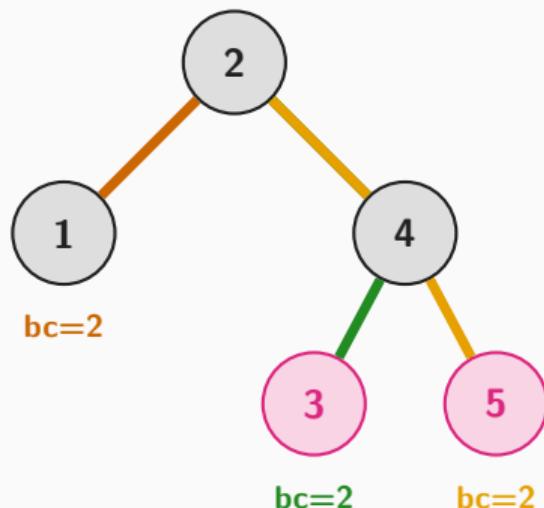
After (node 5 removed)



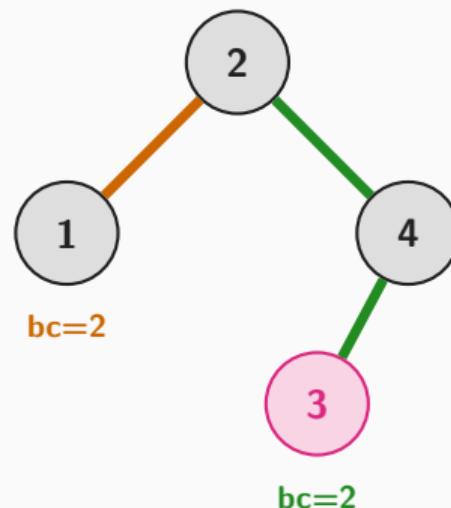
## Case 1: Black-Height Stays the Same

Every path still has  $\text{bc} = 2$  black nodes after removing 5

Before (with node 5)



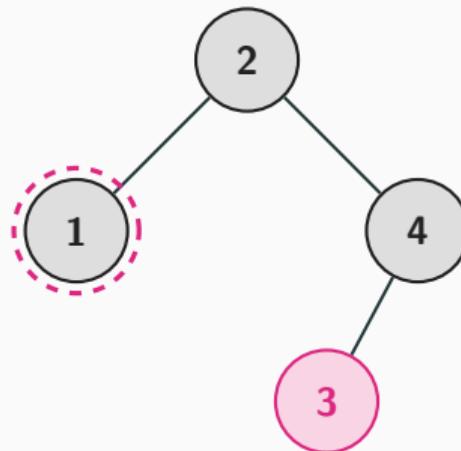
After (node 5 removed)



## Case 2: Deleting a BLACK Node

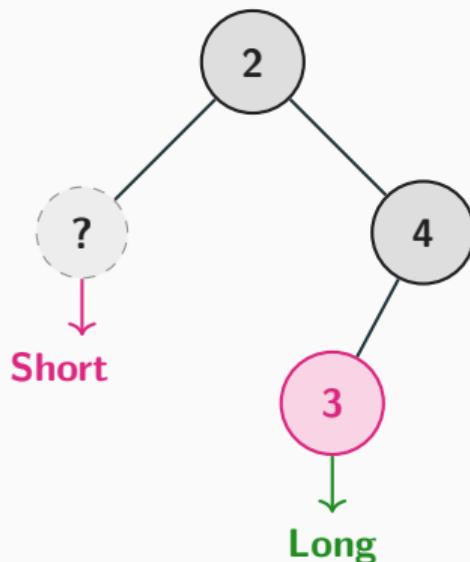
Delete node **1** from the tree

**Before deletion:**



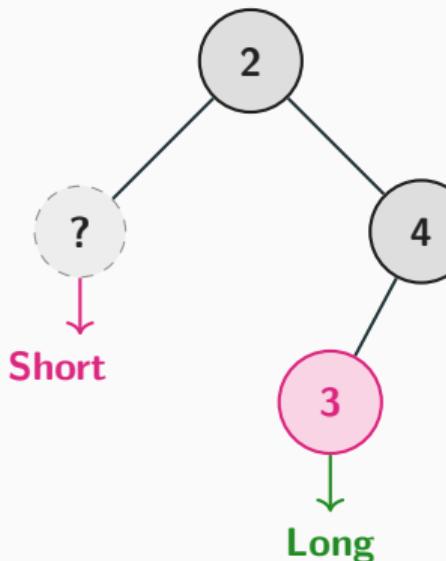
## Case 2: After Deleting the BLACK Node

The node is gone - but now we have a **problem**



## Case 2: After Deleting the BLACK Node

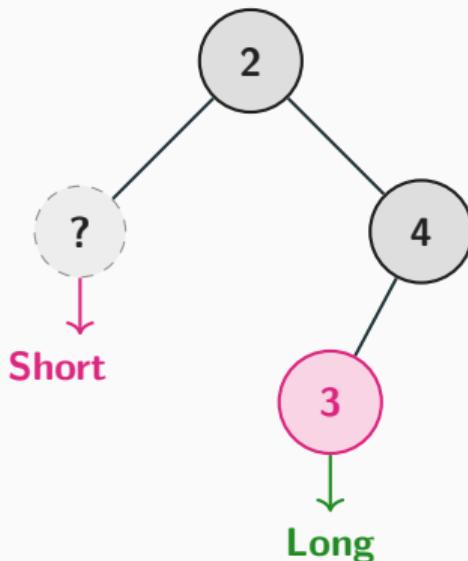
The node is gone - but now we have a **problem**



- Left path is now **shorter**

## Case 2: After Deleting the BLACK Node

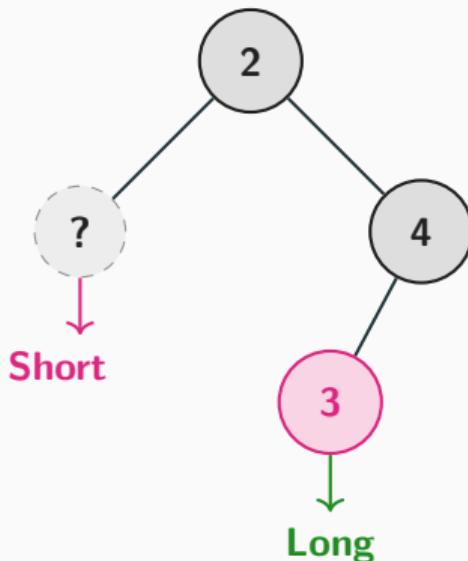
The node is gone - but now we have a **problem**



- Left path is now **shorter**
- Black-height **violated!**

## Case 2: After Deleting the BLACK Node

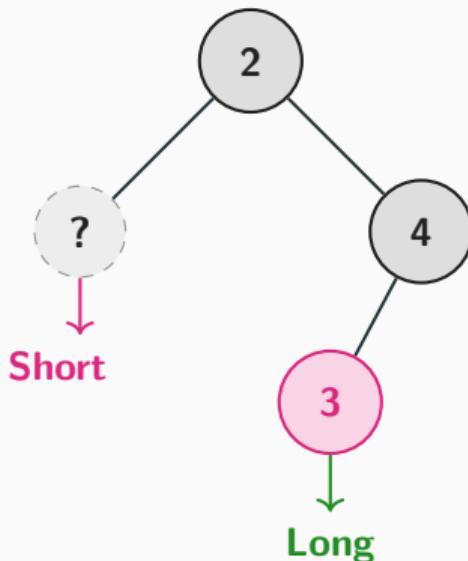
The node is gone - but now we have a **problem**



- Left path is now **shorter**
- Black-height **violated!**
- We call this a  
**“Double-Black” node**

## Case 2: After Deleting the BLACK Node

The node is gone - but now we have a **problem**



- Left path is now **shorter**
- Black-height **violated!**
- We call this a  
**“Double-Black” node**

**IMBALANCED - must fix!**

## Case 2: The Fix-up

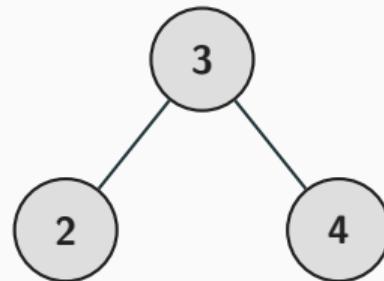
- **Rotate:** Right at 4,  
then left at 2

## Case 2: The Fix-up

- **Rotate:** Right at 4,  
then left at 2
- **Recolor:** Node 3 → Black

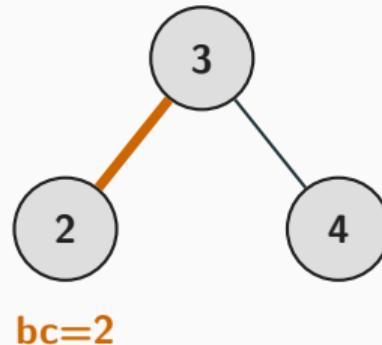
## Case 2: The Fix-up

- **Rotate:** Right at 4,  
then left at 2
- **Recolor:** Node 3 → Black



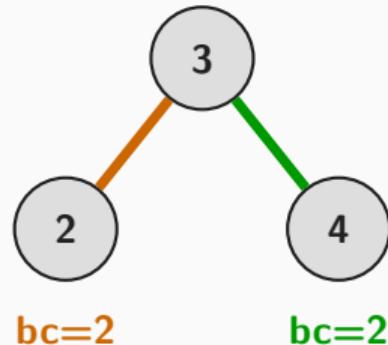
## Case 2: The Fix-up

- **Rotate:** Right at 4,  
then left at 2
- **Recolor:** Node 3 → Black



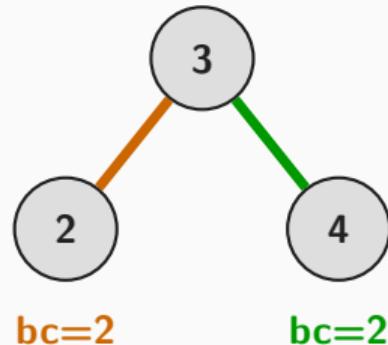
## Case 2: The Fix-up

- **Rotate:** Right at 4,  
then left at 2
- **Recolor:** Node 3 → Black



## Case 2: The Fix-up

- **Rotate:** Right at 4,  
then left at 2
- **Recolor:** Node 3 → Black
- **Tree is balanced!**



All paths:  $bc = 2$  - Black-height restored!

## Fixing Double-Black: 4 Cases

When we have a **Double-Black** node,  
the fix depends on the **sibling's color and children**.

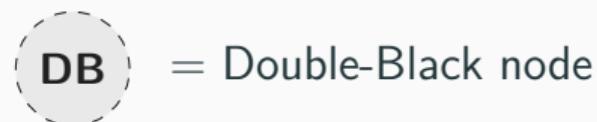
**P** = Parent      **S** = Sibling      **L / R** = S's children

 = Double-Black node

## Fixing Double-Black: 4 Cases

When we have a **Double-Black** node,  
the fix depends on the **sibling's color and children**.

**P** = Parent      **S** = Sibling      **L / R** = S's children

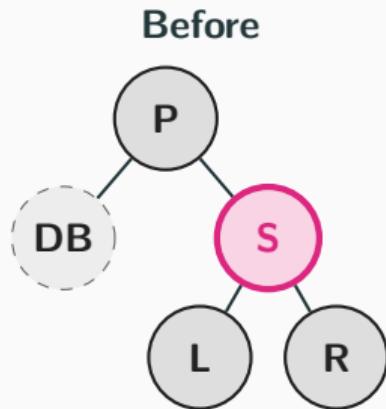


= Double-Black node

**4 cases - let's go through them one by one!**

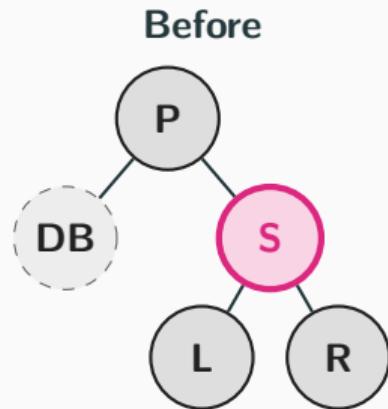
## Fix Case 1 of 4: Sibling is RED

The Sibling S is RED



## Fix Case 1 of 4: Sibling is RED

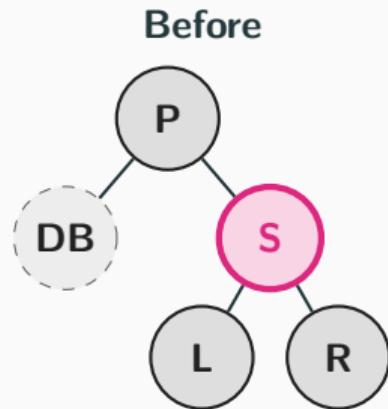
The Sibling S is RED



- **Rotate P to the left**

## Fix Case 1 of 4: Sibling is RED

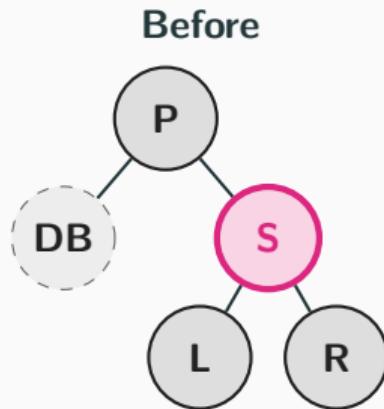
The Sibling S is RED



- **Rotate P to the left**
- **Recolor:** S → Black, P → Red

## Fix Case 1 of 4: Sibling is RED

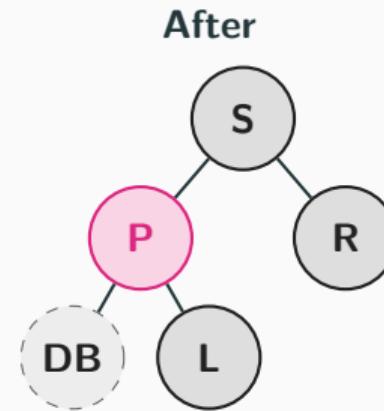
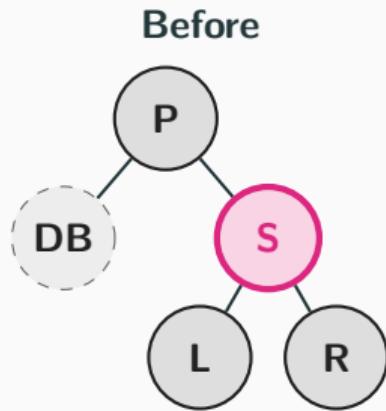
The Sibling S is RED



- **Rotate P to the left**
- **Recolor:** S → Black, P → Red

## Fix Case 1 of 4: Sibling is RED

The Sibling S is RED

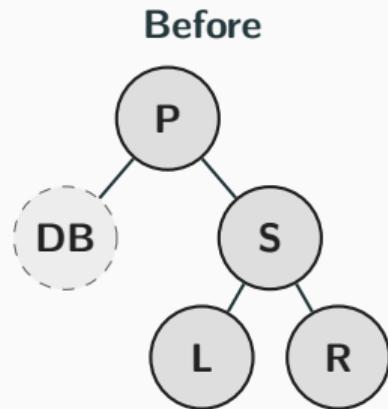


Now apply Case 2, 3, or 4 to DB

- **Rotate P to the left**
- **Recolor:** S → Black, P → Red

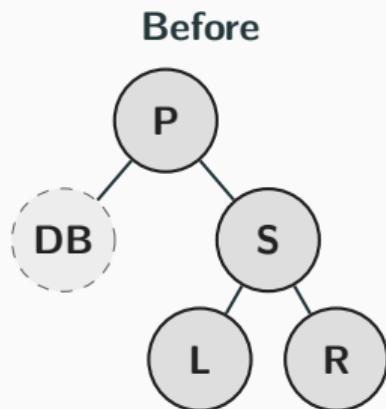
## Fix Case 2 of 4: Sibling & Children All BLACK

S and both children are BLACK



## Fix Case 2 of 4: Sibling & Children All BLACK

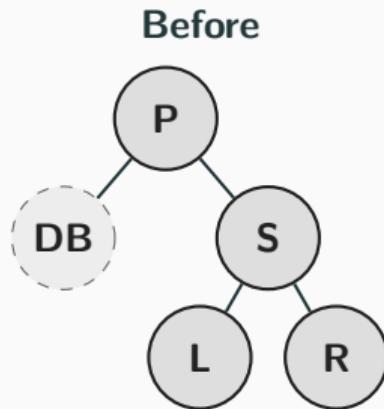
S and both children are BLACK



- Recolor S → Red

## Fix Case 2 of 4: Sibling & Children All BLACK

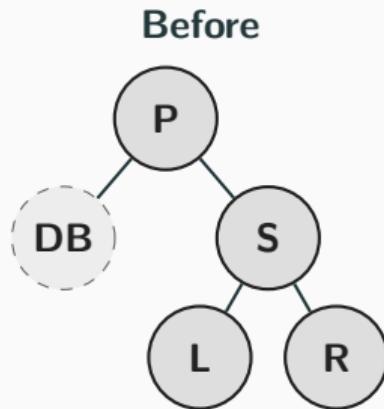
S and both children are BLACK



- Recolor S → Red
- Push the Double-Black up to P

## Fix Case 2 of 4: Sibling & Children All BLACK

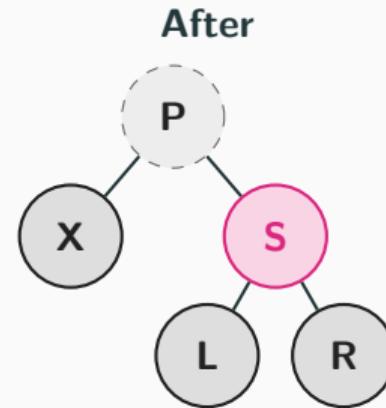
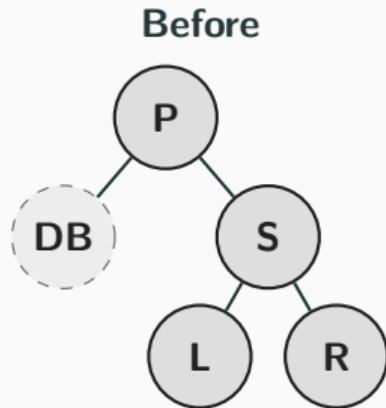
S and both children are BLACK



- Recolor S → Red
- Push the Double-Black up to P

## Fix Case 2 of 4: Sibling & Children All BLACK

S and both children are BLACK

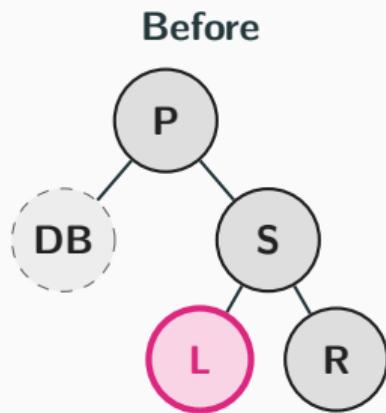


*DB pushed to P — continue fixing*

- Recolor S → Red
- Push the Double-Black up to P

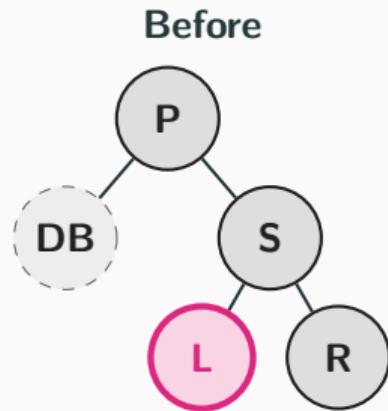
## Fix Case 3 of 4: Sibling's Left Child is RED

S is Black, S's Left child is RED



## Fix Case 3 of 4: Sibling's Left Child is RED

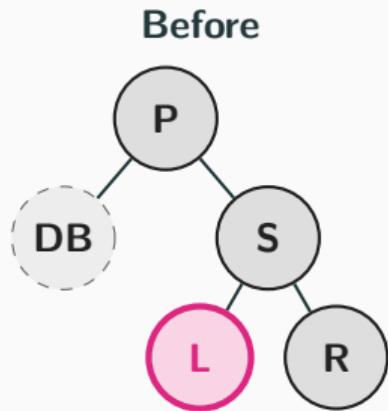
S is Black, S's Left child is RED



- Right-rotate at S, & Swap colors of S and L

## Fix Case 3 of 4: Sibling's Left Child is RED

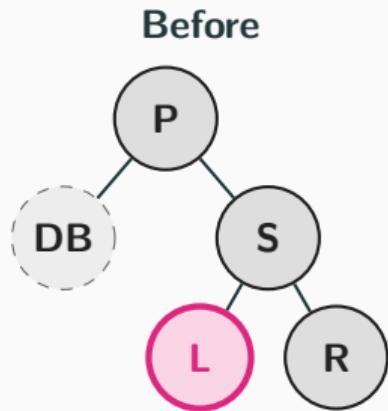
S is Black, S's Left child is RED



- Right-rotate at S, & Swap colors of S and L

## Fix Case 3 of 4: Sibling's Left Child is RED

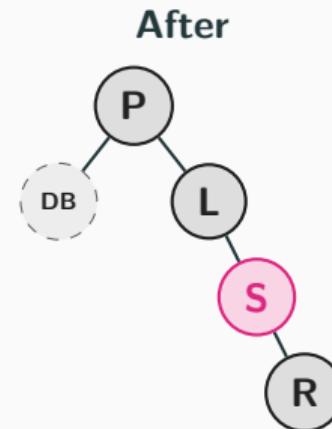
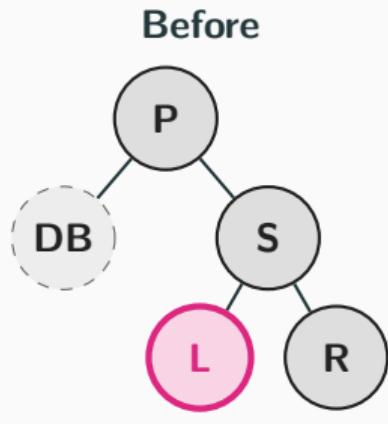
S is Black, S's Left child is RED



- Right-rotate at S, & Swap colors of S and L

## Fix Case 3 of 4: Sibling's Left Child is RED

S is Black, S's Left child is RED

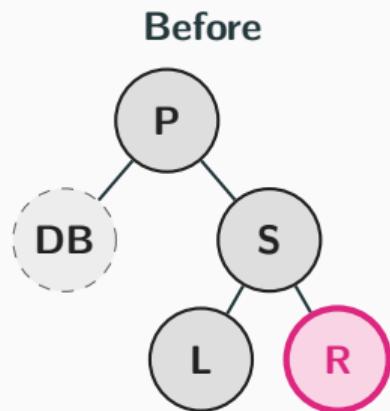


*Now proceed with Case 4*

- Right-rotate at S, & Swap colors of S and L

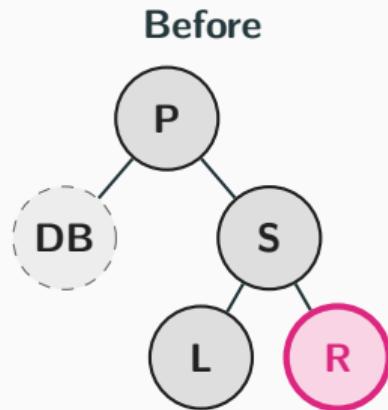
## Fix Case 4 of 4: Sibling's Right Child is RED

S is Black, S's Right child is RED



## Fix Case 4 of 4: Sibling's Right Child is RED

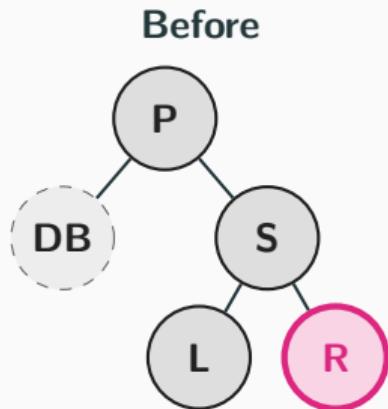
S is Black, S's Right child is RED



- Left-rotate at P

## Fix Case 4 of 4: Sibling's Right Child is RED

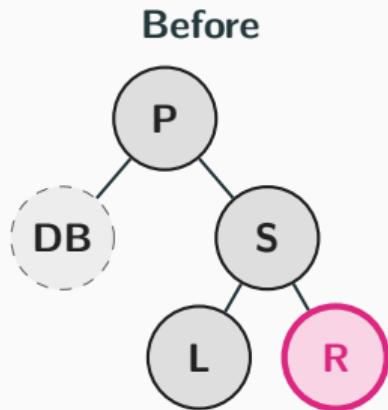
S is Black, S's Right child is RED



- **Left-rotate at P**
- **Recolor R → Black**

## Fix Case 4 of 4: Sibling's Right Child is RED

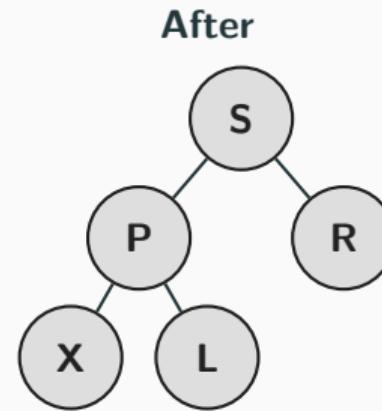
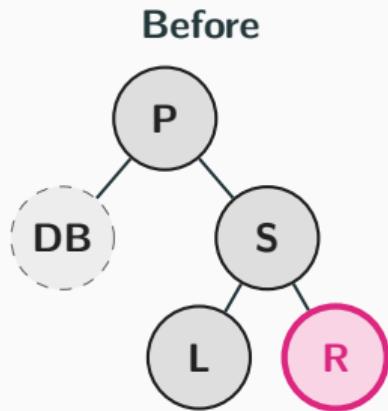
S is Black, S's Right child is RED



- **Left-rotate at P**
- **Recolor R → Black**

## Fix Case 4 of 4: Sibling's Right Child is RED

S is Black, S's Right child is RED

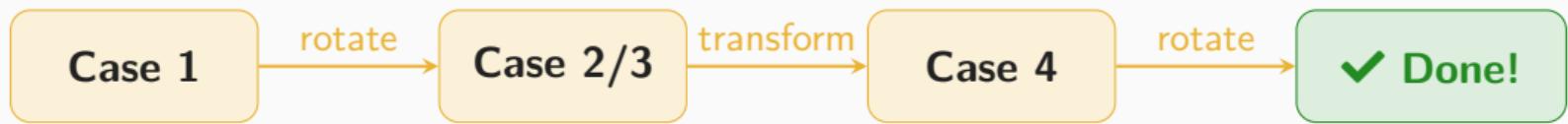


*Double-Black fully resolved!*

- Left-rotate at P
- Recolor R → Black

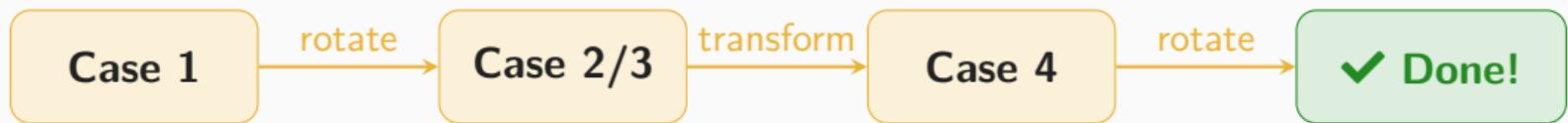
## Summary

The cases form a **chain**:



## Summary

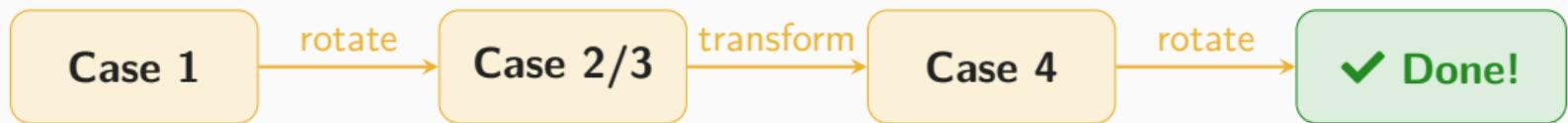
The cases form a **chain**:



**Goal:** Eventually reach Case 4 to fully eliminate Double-Black

## Summary

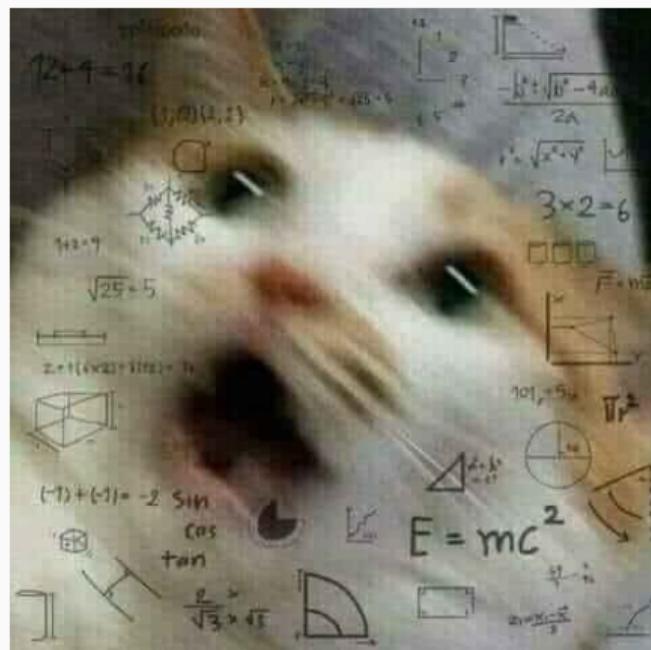
The cases form a **chain**:



**Goal:** Eventually reach Case 4 to fully eliminate Double-Black

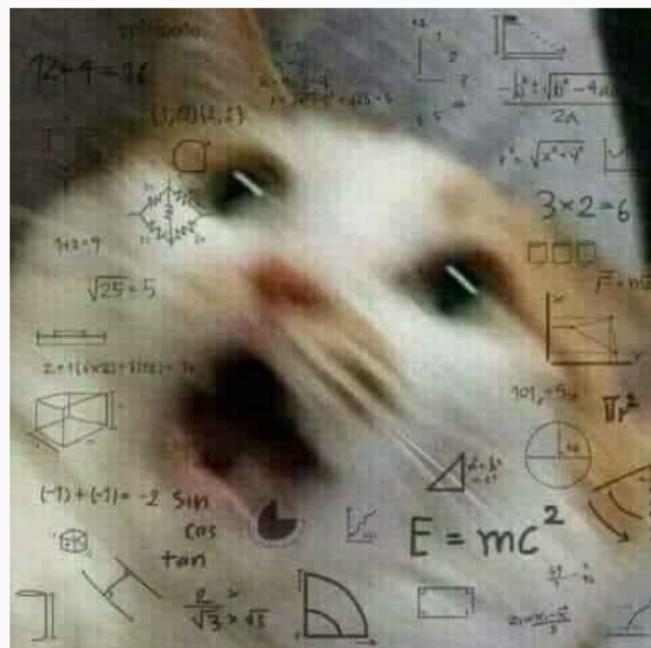
*Case 2 may propagate upward; Cases 1 & 3 always lead to Case 4*

# Too Many Cases?



Confused?

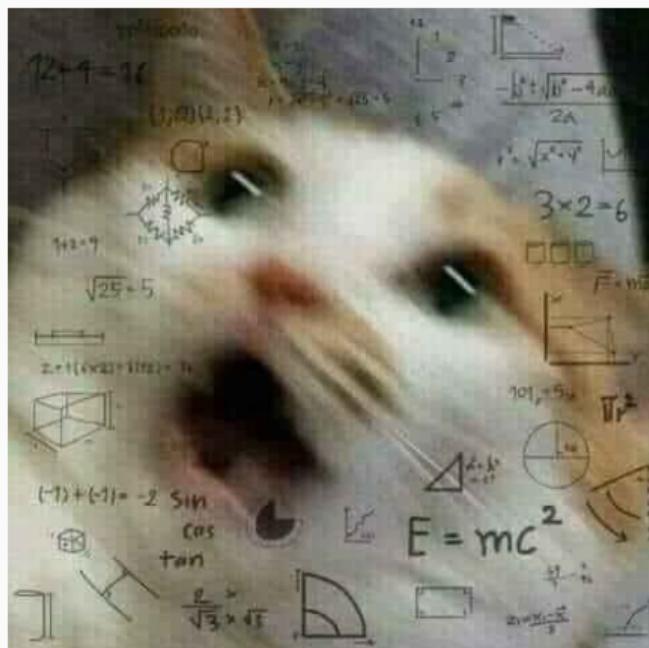
# Too Many Cases?



Confused?

If this felt like **a lot** at once -  
that's because **it is !**

## Too Many Cases?



Confused?

If this felt like **a lot** at once -  
that's because **it is !**

We know deletion is  
complex - and that's **okay!**

## Was All That Worth It?

All that work.

## Was All That Worth It?

All that work.

What did it actually buy us?

## Was All That Worth It?

**All that work.**

What did it actually buy us?

*Let's finally see the payoff.*

## Why Rotations Work: The Magic Behind Balance

**Rotation Complexity:  $O(1)$  Operations**

### Rotation Complexity: $O(1)$ Operations

#### Left Rotation:

- Restructures tree locally

#### Right Rotation:

- Mirror of left rotation

### Rotation Complexity: $O(1)$ Operations

#### Left Rotation:

- Restructures tree locally
- Preserves binary search property

#### Right Rotation:

- Mirror of left rotation
- Same time complexity

## Rotation Complexity: $O(1)$ Operations

### Left Rotation:

- Restructures tree locally
- Preserves binary search property
- Height changes by at most 1

### Right Rotation:

- Mirror of left rotation
- Same time complexity
- Maintains red-black properties

## Rotation Complexity: $O(1)$ Operations

### Left Rotation:

- Restructures tree locally
- Preserves binary search property
- Height changes by at most 1

### Right Rotation:

- Mirror of left rotation
- Same time complexity
- Maintains red-black properties

### Key Insight

Rotations are  $O(1)$  because they only change a constant number of pointers! No tree traversal needed - just pointer gymnastics!

# Why Rotations Work: The Magic Behind Balance

## Rotation Complexity: $O(1)$ Operations

### Left Rotation:

- Restructures tree locally
- Preserves binary search property
- Height changes by at most 1

### Right Rotation:

- Mirror of left rotation
- Same time complexity
- Maintains red-black properties

### Key Insight

Rotations are  $O(1)$  because they only change a constant number of pointers! No tree traversal needed - just pointer gymnastics!



### Height Proof: Why $O(\log n)$ ?

### Height Proof: Why $O(\log n)$ ?

#### Key Insight

A red-black tree with  $n$  internal nodes has height  $\leq 2 \log_2(n + 1)$

### Height Proof: Why $O(\log n)$ ?

#### Key Insight

A red-black tree with  $n$  internal nodes has height  $\leq 2 \log_2(n + 1)$

#### Proof Sketch:

1. Every path from root to leaf has same number of black nodes
2. Red nodes can't have red children
3. At least half nodes on any path are black

# Why Red-Black Trees Work: The Math Behind It

## Height Proof: Why $O(\log n)$ ?

### Key Insight

A red-black tree with  $n$  internal nodes has height  $\leq 2 \log_2(n + 1)$

### Proof Sketch:

1. Every path from root to leaf has same number of black nodes
2. Red nodes can't have red children
3. At least half nodes on any path are black
4. Height  $\leq 2 \times$  black-height

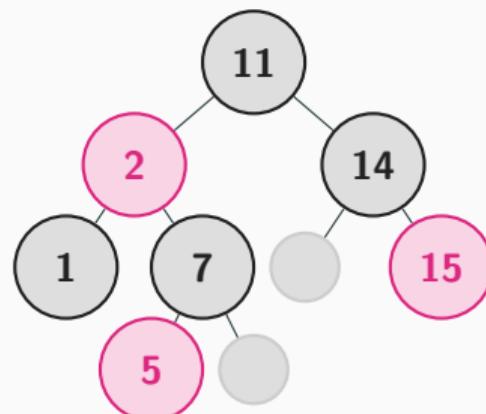
## Height Proof: Why $O(\log n)$ ?

### Key Insight

A red-black tree with  $n$  internal nodes has height  $\leq 2 \log_2(n + 1)$

### Proof Sketch:

1. Every path from root to leaf has same number of black nodes
2. Red nodes can't have red children
3. At least half nodes on any path are black
4. Height  $\leq 2 \times$  black-height



Black height = 2, Total height = 4

## Why Red-Black Trees Are Space Efficient

Space Complexity:  $O(n)$  Memory Usage

### Space Complexity: $O(n)$ Memory Usage

#### Memory Requirements:

- Each node stores: key, color and 2 pointers

#### Comparison:

- AVL trees: height field per node

### Space Complexity: $O(n)$ Memory Usage

#### Memory Requirements:

- Each node stores: key, color and 2 pointers
- Total:  $O(n)$  space

#### Comparison:

- AVL trees: height field per node
- B-trees: multiple keys per node

### Space Complexity: $O(n)$ Memory Usage

#### Memory Requirements:

- Each node stores: key, color and 2 pointers
- Total:  $O(n)$  space
- No extra storage for balance info

#### Comparison:

- AVL trees: height field per node
- B-trees: multiple keys per node
- RBTs: minimal overhead  
**[squirrel-level efficient]**

## Why Red-Black Trees Are Space Efficient

### Space Complexity: $O(n)$ Memory Usage

#### Memory Requirements:

- Each node stores: key, color and 2 pointers
- Total:  $O(n)$  space
- No extra storage for balance info

#### Comparison:

- AVL trees: height field per node
- B-trees: multiple keys per node
- RBTs: minimal overhead  
**[squirrel-level efficient]**

#### Key Insight

Red-black trees achieve  $O(n)$  space with just 1 extra bit per node (the color)!

That's the definition of space-efficient data structures!

50+ Years of Tree Balancing Innovation

### 50+ Years of Tree Balancing Innovation

- 1972: Red-Black Trees (Guibas & Sedgewick) [guibas1972]

### 50+ Years of Tree Balancing Innovation

- **1972:** Red-Black Trees (Guibas & Sedgewick) [guibas1972]
- **2008:** Left-Leaning Red-Black Trees (Sedgewick) [sedgewick2008]

### 50+ Years of Tree Balancing Innovation

- **1972:** Red-Black Trees (Guibas & Sedgewick) [guibas1972]
- **2008:** Left-Leaning Red-Black Trees (Sedgewick) [sedgewick2008]
- **2013:** WAVL Trees (Bronson et al.) [bronson2013]

### 50+ Years of Tree Balancing Innovation

- **1972:** Red-Black Trees (Guibas & Sedgewick) [guibas1972]
- **2008:** Left-Leaning Red-Black Trees (Sedgewick) [sedgewick2008]
- **2013:** WAVL Trees (Bronson et al.) [bronson2013]

**What to do with these trees?**

"Yikes! Trees evolving faster than my code." - Some sad developer

### 50+ Years of Tree Balancing Innovation

- **1972:** Red-Black Trees (Guibas & Sedgewick) [guibas1972]
- **2008:** Left-Leaning Red-Black Trees (Sedgewick) [sedgewick2008]
- **2013:** WAVL Trees (Bronson et al.) [bronson2013]

**What to do with these trees?**

"Yikes! Trees evolving faster than my code." - Some sad developer

**Fun fact:** Robert Sedgewick (co-inventor of RBT) later said: "*I prefer left-leaning red-black trees now - they're simpler!*"

### Red-Black Trees in Modern Tech

# Red-Black Trees in Modern Tech

### Traditional Uses:

- CPU scheduling algorithms

# Red-Black Trees in Modern Tech

### Traditional Uses:

- CPU scheduling algorithms
- Memory management

# Red-Black Trees in Modern Tech

### Traditional Uses:

- CPU scheduling algorithms
- Memory management
- Network routing tables

# Red-Black Trees in Modern Tech

### Traditional Uses:

- CPU scheduling algorithms
- Memory management
- Network routing tables
- Game engines (spatial indexing)

### Red-Black Trees in Modern Tech

#### Traditional Uses:

- CPU scheduling algorithms
- Memory management
- Network routing tables
- Game engines (spatial indexing)

#### Modern Applications:

# Red-Black Trees in Modern Tech

### Traditional Uses:

- CPU scheduling algorithms
- Memory management
- Network routing tables
- Game engines (spatial indexing)

### Modern Applications:

- ML indexing, cloud storage, blockchain, AI pathfinding!



# Red-Black Trees in Modern Tech

### Traditional Uses:

- CPU scheduling algorithms
- Memory management
- Network routing tables
- Game engines (spatial indexing)



### Modern Applications:

- ML indexing, cloud storage, blockchain, AI pathfinding!
- RBTs: Keanu Reeves of data structures - always reliable!



### Why Choose Red-Black Trees?

## The Ultimate Showdown: RBT vs Other Trees

### Why Choose Red-Black Trees?

Tree Type	Search	Insert	Delete	Space
Red-Black	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
B-Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

## The Ultimate Showdown: RBT vs Other Trees

### Why Choose Red-Black Trees?

Tree Type	Search	Insert	Delete	Space
Red-Black	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
B-Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

## Red-Black Advantages: The Meme Edition

Why Red-Black Trees Are the Perfect Choice

# Red-Black Advantages: The Meme Edition

## Why Red-Black Trees Are the Perfect Choice



### The Goldilocks Solution

Not too fast, not too slow,  
just right!

# Red-Black Advantages: The Meme Edition

## Why Red-Black Trees Are the Perfect Choice



### The Goldilocks Solution

Not too fast, not too slow,  
just right!



### Rock-Solid Guarantees

Faster insertions than AVL,  
simpler than B-trees!

# Red-Black Advantages: The Meme Edition

## Why Red-Black Trees Are the Perfect Choice



### The Goldilocks Solution

Not too fast, not too slow,  
just right!



### Rock-Solid Guarantees

Faster insertions than AVL,  
simpler than B-trees!



### Perfect Balance

Like coffee - balanced and  
reliable!

### The Real Story Behind the Scenes

### The Real Story Behind the Scenes

Tech Giants & RBTs:

- **Google:** Uses RBTs in MapReduce

### The Real Story Behind the Scenes

#### Tech Giants & RBTs:

- **Google:** Uses RBTs in MapReduce
- **Facebook:** News feed ranking

#### Secret Sauce

Many companies use hybrid approaches - RBTs for small datasets, B-trees for large ones. It's complicated... but mostly red-black trees!

### The Real Story Behind the Scenes

#### Tech Giants & RBTs:

- **Google:** Uses RBTs in MapReduce
- **Facebook:** News feed ranking
- **Amazon:** Product recommendations

#### Secret Sauce

Many companies use hybrid approaches - RBTs for small datasets, B-trees for large ones. It's complicated... but mostly red-black trees!

### The Real Story Behind the Scenes

#### Tech Giants & RBTs:

- **Google:** Uses RBTs in MapReduce
- **Facebook:** News feed ranking
- **Amazon:** Product recommendations
- **Netflix:** Content delivery networks

#### Secret Sauce

Many companies use hybrid approaches - RBTs for small datasets, B-trees for large ones. It's complicated... but mostly red-black trees!

### The Real Story Behind the Scenes

#### Tech Giants & RBTs:

- **Google:** Uses RBTs in MapReduce
- **Facebook:** News feed ranking
- **Amazon:** Product recommendations
- **Netflix:** Content delivery networks
- **Microsoft:** Windows kernel

#### Secret Sauce

Many companies use hybrid approaches - RBTs for small datasets, B-trees for large ones. It's complicated... but mostly red-black trees!

### The Real Story Behind the Scenes

#### Tech Giants & RBTs:

- **Google:** Uses RBTs in MapReduce
- **Facebook:** News feed ranking
- **Amazon:** Product recommendations
- **Netflix:** Content delivery networks
- **Microsoft:** Windows kernel

#### Secret Sauce

Many companies use hybrid approaches - RBTs for small datasets, B-trees for large ones. It's complicated... but mostly red-black trees!

# The Real Story Behind the Scenes

### Tech Giants & RBTs:

- **Google:** Uses RBTs in MapReduce
- **Facebook:** News feed ranking
- **Amazon:** Product recommendations
- **Netflix:** Content delivery networks
- **Microsoft:** Windows kernel

### Secret Sauce

Many companies use hybrid approaches - RBTs for small datasets, B-trees for large ones. It's complicated... but mostly red-black trees!



### Next Generation Data Structures

### Next Generation Data Structures

#### Emerging Trends:

- **Persistent Trees** (functional programming)

## Next Generation Data Structures

### Emerging Trends:

- **Persistent Trees** (functional programming)
- **Concurrent RBTs** (multi-threading)

## Next Generation Data Structures

### Emerging Trends:

- **Persistent Trees** (functional programming)
- **Concurrent RBTs** (multi-threading)
- **GPU-accelerated trees**

## Next Generation Data Structures

### Emerging Trends:

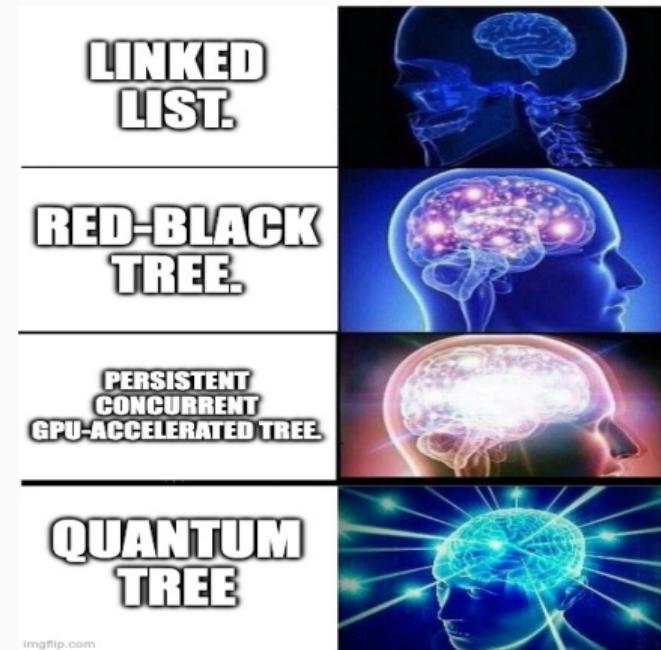
- **Persistent Trees** (functional programming)
- **Concurrent RBTs** (multi-threading)
- **GPU-accelerated trees**
- **Quantum-inspired data structures** - because we're not sure what they do, but they sound cool!

# The Future: Where Are Trees Heading?

## Next Generation Data Structures

### Emerging Trends:

- **Persistent Trees** (functional programming)
- **Concurrent RBTs** (multi-threading)
- **GPU-accelerated trees**
- **Quantum-inspired data structures** - because we're not sure what they do, but they sound cool!



**So There You Have It!**

## Red-Black Trees: The Unsung Heroes

### Red-Black Trees: The Unsung Heroes

- Mathematical elegance

### Red-Black Trees: The Unsung Heroes

- Mathematical elegance
- Practical performance

### Red-Black Trees: The Unsung Heroes

- Mathematical elegance
- Practical performance
- Everywhere in computing

### Red-Black Trees: The Unsung Heroes

- Mathematical elegance
- Practical performance
- Everywhere in computing

#### Hero Status

Red-Black Trees: Not all heroes wear capes!

# Red-Black Trees: The Unsung Heroes

- Mathematical elegance
- Practical performance
- Everywhere in computing

### Hero Status

Red-Black Trees: Not all heroes wear capes!

### Remember

The next time your code runs in  $O(\log n)$  time...

Thank a red-black tree! ❤

# Thanks for listening!

# Thanks for listening!

One Last thing before we go...

# Thanks for listening!

One Last thing before we go...

