# Red-Black Trees

Why Even the Inventor Moved On...

Your Name

February 19, 2026

# We All Love to Sort Things!

- Organizing bookshelves

# We All Love to Sort Things!

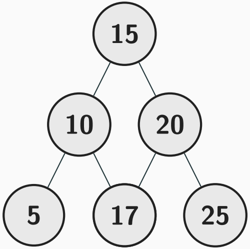
- Organizing bookshelves
- Arranging files on computers

# We All Love to Sort Things!

- Organizing bookshelves
- Arranging files on computers
- Putting groceries in order

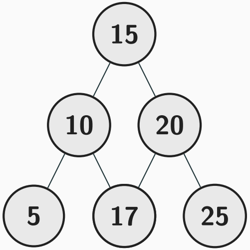
# We All Love to Sort Things!

- Organizing bookshelves
- Arranging files on computers
- Putting groceries in order
- **What's a really good way?** 🌲



# We All Love to Sort Things!

- Organizing bookshelves
- Arranging files on computers
- Putting groceries in order
- **What's a really good way?** 🌲
- **TREES!** Specifically Binary Trees




**Binary Trees are Awesome!**

# Why Binary Trees Are Good

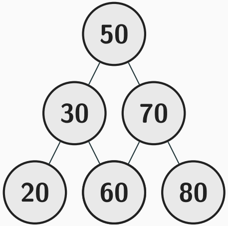
## Beautiful Structure

- Everything has its place
- Search: $O(\log n)$
- Insert: $O(\log n)$
- Delete: $O(\log n)$

# Why Binary Trees Are Good

### Beautiful Structure

- Everything has its place
- Search: $O(\log n)$
- Insert: $O(\log n)$
- Delete: $O(\log n)$




### The Magic

Logarithmic time = **Sports car performance!**

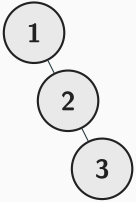
# Sadly, Sometimes It Goes Wrong...

What happens when we insert: 1, 2, 3, 4, 5?

# Sadly, Sometimes It Goes Wrong...

What happens when we insert: 1, 2, 3, 4, 5?

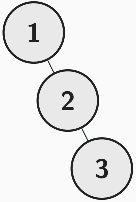
**After 1, 2, 3**



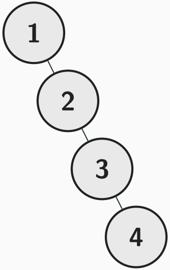
# Sadly, Sometimes It Goes Wrong...

What happens when we insert: 1, 2, 3, 4, 5?

**After 1, 2, 3**

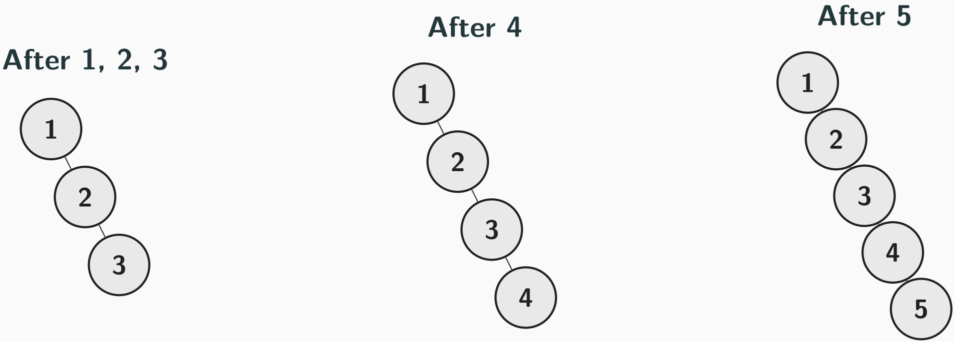



**After 4**



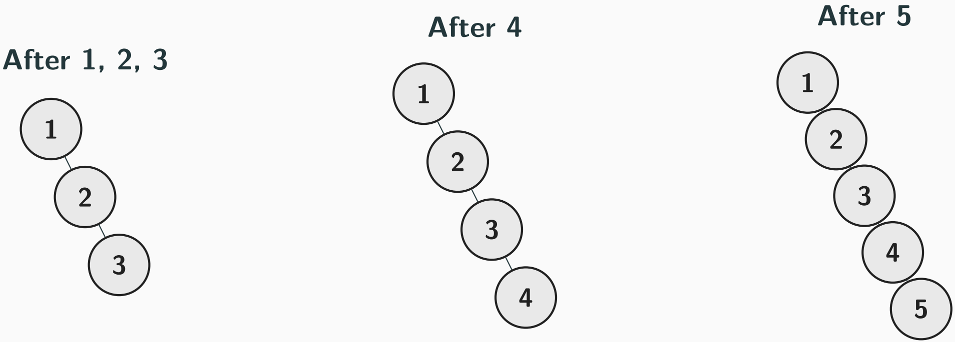
# Sadly, Sometimes It Goes Wrong...

What happens when we insert: 1, 2, 3, 4, 5?



## Sadly, Sometimes It Goes Wrong...

What happens when we insert: 1, 2, 3, 4, 5?

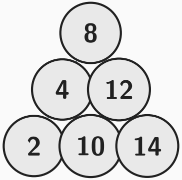



**Our tree became a... LINKED LIST!** Damn!

# From Sports Car to Bicycle

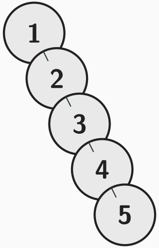
**Before (Balanced)**

- Search: $O(\log n)$
- Insert: $O(\log n)$
- Delete: $O(\log n)$




**After (Degenerate)**

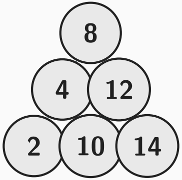
- Search: $O(n)$
- Insert: $O(n)$
- Delete: $O(n)$



# From Sports Car to Bicycle

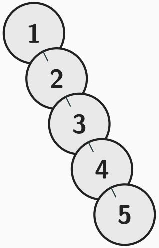
**Before (Balanced)**

- Search: $O(\log n)$
- Insert: $O(\log n)$
- Delete: $O(\log n)$




**After (Degenerate)**

- Search: $O(n)$
- Insert: $O(n)$
- Delete: $O(n)$




Not Great!

# We Need Almost Balanced Trees

We want to keep $O(\log n)$ operations **even in the worst case!**

# We Need Almost Balanced Trees

We want to keep $O(\log n)$ operations **even in the worst case!**

**Popular Solutions**

- AVL Trees

# We Need Almost Balanced Trees

We want to keep $O(\log n)$ operations **even in the worst case!**

**Popular Solutions**

- AVL Trees
- **Red-Black Trees** ★

# We Need Almost Balanced Trees

We want to keep $O(\log n)$ operations **even in the worst case!**

**Popular Solutions**

- AVL Trees
- **Red-Black Trees** ★
- B-Trees

# We Need Almost Balanced Trees

We want to keep $O(\log n)$ operations **even in the worst case!**

**Popular Solutions**

- AVL Trees
- **Red-Black Trees** ★
- B-Trees
- Splay Trees

# We Need Almost Balanced Trees

We want to keep $O(\log n)$ operations **even in the worst case!**

**Popular Solutions**

- AVL Trees
- **Red-Black Trees** ★
- B-Trees
- Splay Trees

**Red-Black Trees are used EVERYWHERE!**

# Enter: Red-Black Trees

- Around since the 1970s

# Enter: Red-Black Trees

- Around since the 1970s
- Used in Linux kernels

# Enter: Red-Black Trees

- Around since the 1970s
- Used in Linux kernels
- Java's TreeMap and TreeSet

# Enter: Red-Black Trees

- Around since the 1970s
- Used in Linux kernels
- Java's TreeMap and TreeSet
- Databases and file systems

# Enter: Red-Black Trees

- Around since the 1970s
- Used in Linux kernels
- Java's TreeMap and TreeSet
- Databases and file systems

**Fun Fact**

Even the original inventor **didn't mention** Red-Black Trees in his main DSA book!

He introduced *Left-Leaning Red-Black Trees* instead.

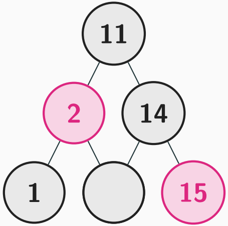
😃

# Enter: Red-Black Trees

- Around since the 1970s
- Used in Linux kernels
- Java's TreeMap and TreeSet
- Databases and file systems

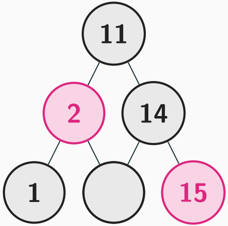
**Fun Fact**

Even the original inventor **didn't mention** Red-Black Trees in his main DSA book!
He introduced *Left-Leaning Red-Black Trees* instead.
😀



# Enter: Red-Black Trees

- Around since the 1970s
- Used in Linux kernels
- Java's TreeMap and TreeSet
- Databases and file systems




**Fun Fact**

Even the original inventor **didn't mention** Red-Black Trees in his main DSA book!
He introduced *Left-Leaning Red-Black Trees* instead.
😀

**Let's get started, shall we?**

## Everything Has Unique Properties

Fundamental properties are better than mathematical definitions!

## Everything Has Unique Properties

Fundamental properties are better than mathematical definitions!

**Why Properties?**

- Easier to understand

# Everything Has Unique Properties

Fundamental properties are better than mathematical definitions!

**Why Properties?**

- Easier to understand
- Audience feels good

# Everything Has Unique Properties

Fundamental properties are better than mathematical definitions!

**Why Properties?**

- Easier to understand
- Audience feels good
- Been around for ages!

# Everything Has Unique Properties

Fundamental properties are better than mathematical definitions!

**Why Properties?**

- Easier to understand
- Audience feels good
- Been around for ages!

**Red-Black Trees have 5 properties**

Let's see them one by one...

# The Five Properties

1. Every node is either Red or Black

# The Five Properties

1. Every node is either Red or Black
2. The **root** is always Black

# The Five Properties

1. Every node is either Red or Black
2. The **root** is always Black
3. All **leaves** (NIL) are Black

# The Five Properties

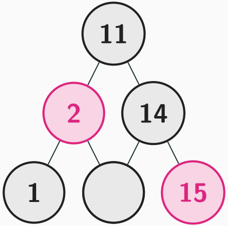
1. Every node is either Red or Black
2. The **root** is always Black
3. All **leaves** (NIL) are Black
4. No two Red nodes can be adjacent

# The Five Properties

1. Every node is either Red or Black
2. The **root** is always Black
3. All **leaves** (NIL) are Black
4. No two Red nodes can be adjacent
5. Equal black height on all paths

## The Five Properties

1. Every node is either Red or Black
2. The **root** is always Black
3. All **leaves** (NIL) are Black
4. No two Red nodes can be adjacent
5. Equal black height on all paths



# Black Height: The Core Property

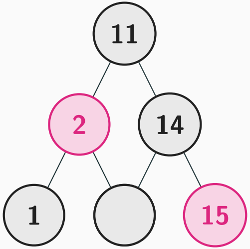
**Definition**

The **black-height** of a node is the number of black nodes on any path from that node to a leaf (not counting the node itself).

# Black Height: The Core Property

## Definition

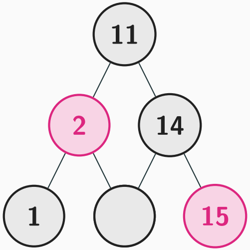
The **black-height** of a node is the number of black nodes on any path from that node to a leaf (not counting the node itself).



# Black Height: The Core Property

## Definition

The **black-height** of a node is the number of black nodes on any path from that node to a leaf (not counting the node itself).




## Why This Matters

This property helps us understand the **balance** of the tree!

# The Amazing Height Bound

Because of black height, we can prove:

# The Amazing Height Bound

Because of black height, we can prove:

**Theorem**

*A Red-Black Tree with $n$ nodes has height $h \leq 2\log_2(n+1)$*

## The Amazing Height Bound

Because of black height, we can prove:

**Theorem**

*A Red-Black Tree with n nodes has height* $h \leq 2\log_2(n+1)$

**What This Means**

- Even worst case is logarithmic!
- Never more than twice as tall as perfect tree
- $O(\log n)$ performance guaranteed

# The Amazing Height Bound

Because of black height, we can prove:

**Theorem**

*A Red-Black Tree with $n$ nodes has height* $h \leq 2\log_2(n+1)$

**What This Means**

- Even worst case is logarithmic!
- Never more than twice as tall as perfect tree
- $O(\log n)$ performance guaranteed

**Perfect Tree:** height $= \log_2(n+1)$

**RBT Tree:** height $\leq 2 \times \log_2(n+1)$

✔ Still Awesome!

# The Amazing Height Bound

Because of black height, we can prove:

**Theorem**

*A Red-Black Tree with $n$ nodes has height* $h \leq 2\log_2(n+1)$

**What This Means**

- Even worst case is logarithmic!
- Never more than twice as tall as perfect tree
- $O(\log n)$ performance guaranteed

**Perfect Tree:** height $= \log_2(n+1)$

**RBT Tree:** height $\leq 2 \times \log_2(n+1)$

✔ Still Awesome!

*(I skipped the detailed proof - nobody wants to sit through that!)*

# Wait, We're Not Done Yet!

We haven't talked about **insertion** or **deletion** yet!

# Wait, We're Not Done Yet!

We haven't talked about **insertion** or **deletion** yet!

**Warning: Shit's about to get real**

*(Also tough)*

## Wait, We're Not Done Yet!

We haven't talked about **insertion** or **deletion** yet!

**Warning: Shit's about to get real**

*(Also tough)*

**How do we insert something?**

# Wait, We're Not Done Yet!

We haven't talked about **insertion** or **deletion** yet!

**Warning: Shit's about to get real**

*(Also tough)*

**How do we insert something?**

# Insertion: The Process

1. Insert like a normal BST

# Insertion: The Process

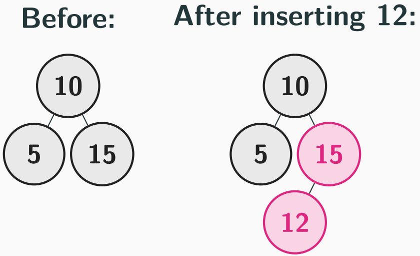
1. Insert like a normal BST
2. Color the new node **RED**

# Insertion: The Process

1. Insert like a normal BST
2. Color the new node **RED**
3. Fix any violations

# Insertion: The Process

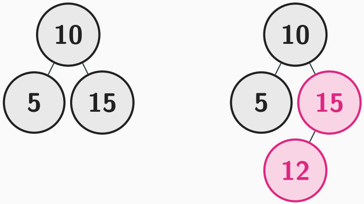
1. Insert like a normal BST
2. Color the new node **RED**
3. Fix any violations



# Insertion: The Process

1. Insert like a normal BST
2. Color the new node **RED**
3. Fix any violations

**Before:** **After inserting 12:**




**Problem!**

Two adjacent red nodes — Property 4 violated!

# Fixing Violations

## Uncle is RED

- Easy case!
- Just recolor
- Flip parent, uncle & grandparent

# Fixing Violations

### Uncle is RED

- Easy case!
- Just recolor
- Flip parent, uncle & grandparent

### Uncle is BLACK

- Hard case!
- Need rotations
- Left rotate, right rotate
- Sometimes both!

# Fixing Violations

**Uncle is RED**

- Easy case!
- Just recolor
- Flip parent, uncle & grandparent

**Uncle is BLACK**

- Hard case!
- Need rotations
- Left rotate, right rotate
- Sometimes both!

This is where it gets tricky!

But it keeps the tree balanced!

# 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!

**Regular BST**

Became a linked list

Height $= 5$

$O(n)$ operations ☹

# Remember This Problem?

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

But this time in a Red-Black Tree!

**Regular BST**

Became a linked list

Height $= 5$

$O(n)$ operations 😢

**Red-Black Tree**

Let's see what happens...

Will it stay balanced?

🪄 Stay tuned!

## Remember This Problem?

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

But this time in a Red-Black Tree!

**Regular BST**

Became a linked list

Height $= 5$

$O(n)$ operations 😢

**Red-Black Tree**

Let's see what happens...

Will it stay balanced?

🪄 Stay tuned!

**Watch the magic happen!**

## Simulation — Step 1 of 5: Insert 1


- First node is always the root

# Simulation — Step 1 of 5: Insert 1


- First node is always the root
- Color it **BLACK**

# Simulation — Step 1 of 5: Insert 1

1 2 3 4 5 Inserting 1

- First node is always the root
- Color it **BLACK**
- Property 2: Root must be black ✔


**All properties satisfied!**

Height $= 1$ Black-height $= 1$

# Simulation — Step 2 of 5: Insert 2

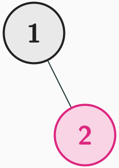

Inserting 2

- Insert as right child of 1

# Simulation — Step 2 of 5: Insert 2


Inserting 2

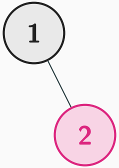
- Insert as right child of 1
- Color it **RED**



# Simulation — Step 2 of 5: Insert 2


Inserting 2

- Insert as right child of 1
- Color it **RED**
- Parent is **BLACK** $\Rightarrow$ no violation!




**Still balanced!**

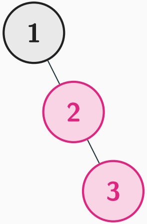
Height $= 2$ Black-height $= 1$

# Simulation — Step 3 of 5: Insert 3 (Fix needed!)

✓ ✓ 3 4 5

Inserting 3 — rotation required

**After Insert**

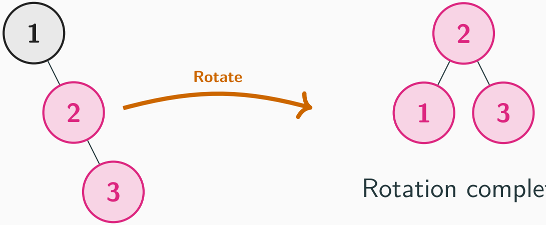



⚠ VIOLATION

# Simulation — Step 3 of 5: Insert 3 (Fix needed!)

✓ ✓ 3 4 5 Inserting 3 — rotation required

**After Insert** | **Left-Rotate at 1**

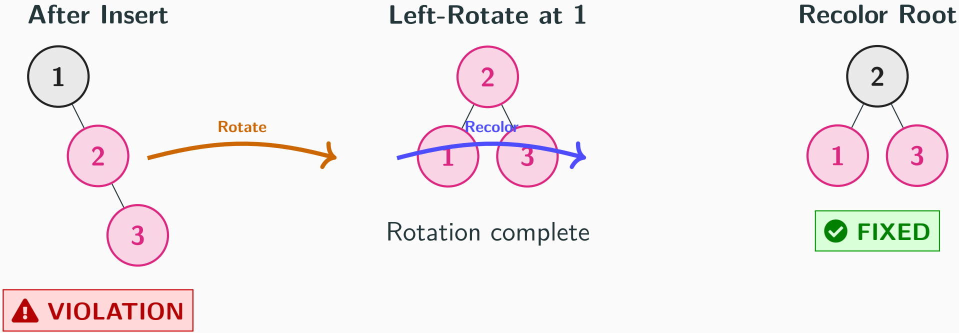



⚠ **VIOLATION**

Rotation complete

## Simulation — Step 3 of 5: Insert 3 (Fix needed!)

Inserting 3 — rotation required


**Balanced!**

Height = 2 (BST would be 3)

# Simulation — Step 4 of 5: Insert 4

Inserting 4 — uncle recolor

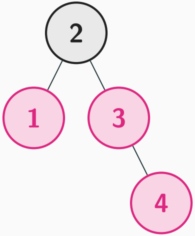
- Insert as right child of 3

# Simulation — Step 4 of 5: Insert 4


Inserting 4 — uncle recolor

- Insert as right child of 3
- Color it **RED**

**After Insert**

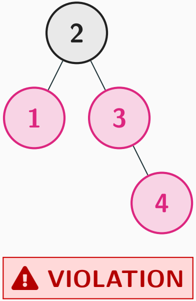


# Simulation — Step 4 of 5: Insert 4


Inserting 4 — uncle recolor

- Insert as right child of 3
- Color it **RED**
- Parent (3) is red — violation!

**After Insert**

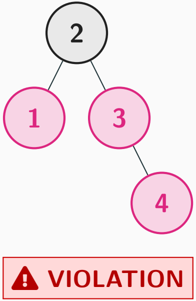


# Simulation — Step 4 of 5: Insert 4


Inserting 4 — uncle recolor

- Insert as right child of 3
- Color it **RED**
- Parent (3) is red — violation!
- Uncle (1) is also red

**After Insert**

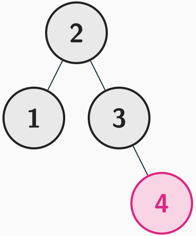


# Simulation — Step 4 of 5: Insert 4

✓ ✓ ✓ 4 5 Inserting 4 — uncle recolor

- Insert as right child of 3
- Color it **RED**
- Parent (3) is red — violation!
- Uncle (1) is also red
- Recolor: flip parent, uncle & grandparent

**After Recolor**




FIXED

**Still balanced!**

Height $= 3$ Black-height preserved

# Simulation — Step 5 of 5: Insert 5 (Final Fix!)


Inserting 5 — rotation + recolor

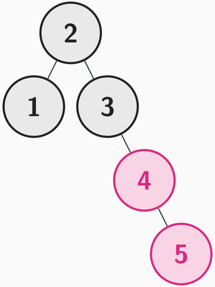
- Insert as right child of 4

## Simulation — Step 5 of 5: Insert 5 (Final Fix!)


Inserting 5 — rotation + recolor

- Insert as right child of 4
- Color it **RED**

**Before Fix**

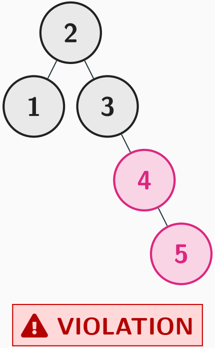


## Simulation — Step 5 of 5: Insert 5 (Final Fix!)

✓ ✓ ✓ ✓ 5 Inserting 5 — rotation + recolor

- Insert as right child of 4
- Color it **RED**
- Parent (4) is red — violation!

**Before Fix**

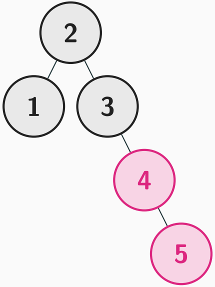


# Simulation — Step 5 of 5: Insert 5 (Final Fix!)


Inserting 5 — rotation + recolor

- Insert as right child of 4
- Color it **RED**
- Parent (4) is red — violation!
- Uncle is **BLACK** (NIL)

**Before Fix**



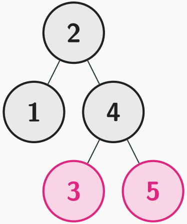

⚠ VIOLATION

# Simulation — Step 5 of 5: Insert 5 (Final Fix!)

Inserting 5 — rotation + recolor

- Insert as right child of 4
- Color it **RED**
- Parent (4) is red — violation!
- Uncle is **BLACK** (NIL)
- Left-rotate at 3, then recolor

**Final Tree**

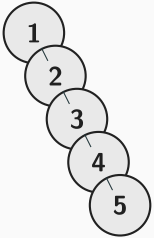



**PERFECT**

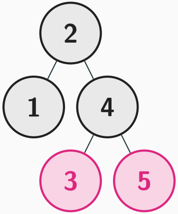
# BST vs. Red-Black Tree

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

**Regular BST**




**Red-Black Tree**




**Bad**

Height $= 5$ ▪ Degenerate ▪ $O(n)$ ops

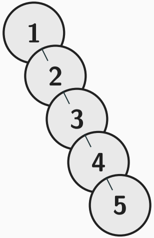
**Excellent!**

Height $= 3$ ▪ Balanced ▪ $O(\log n)$ ops

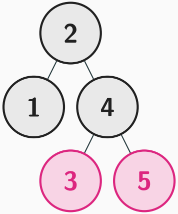
# BST vs. Red-Black Tree

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

### Regular BST




### Red-Black Tree




**Bad**

Height $= 5$ ▪ Degenerate ▪ $O(n)$ ops

**Excellent!**

Height $= 3$ ▪ Balanced ▪ $O(\log n)$ ops

# Deletion: Even More Fun!

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

## Deletion: Even More Fun!

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

**Deleting a RED node**

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

# Deletion: Even More Fun!

## 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: Even More Fun!

## Deletion is even more... interesting! ☻

### Deleting a RED node

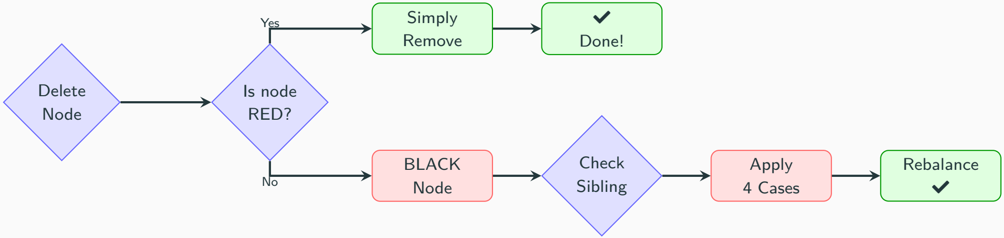
- No problem!
- Just remove it
- Properties still hold

### Deleting a BLACK node

- Oh boy...
- Black height changes!
- Need "double black" fix
- Complex cases

## Let's see both cases...

# Deletion Decision Flowchart

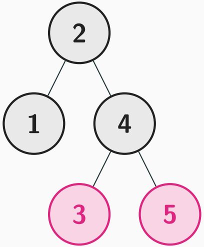



**Top path (RED)** = easy **Bottom path (BLACK)** = complex

# Case 1: Deleting a RED Node (Easy!)

Delete **5** from our tree

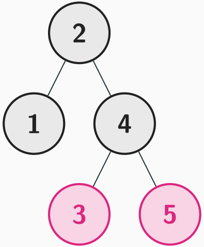
**Before**



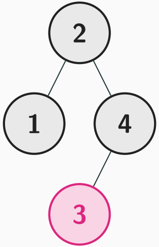
# Case 1: Deleting a RED Node (Easy!)

Delete **5** from our tree

**Before**



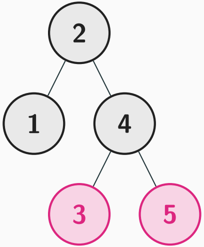

**After**



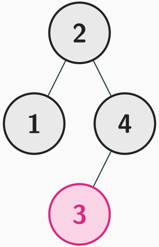
## Case 1: Deleting a RED Node (Easy!)

Delete **5** from our tree

**Before**




**After**



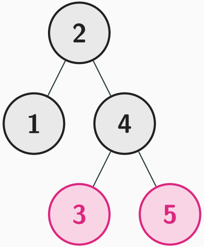

**Why It's Easy**

- Node 5 is RED and a leaf

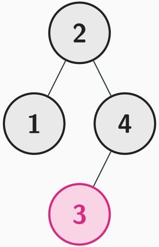
# Case 1: Deleting a RED Node (Easy!)

Delete **5** from our tree

**Before**




**After**

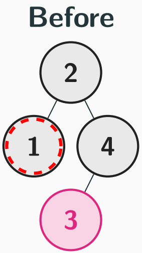



**Why It's Easy**

- Node 5 is RED and a leaf

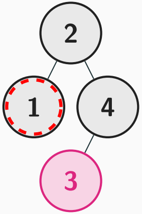
# Case 2: Deleting a BLACK Node (Uh oh...)

Delete **1** from our tree

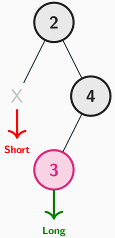


Delete **1** from our tree

**Before**




**After Delete**

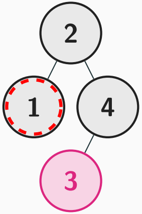



✕ IMBALANCED

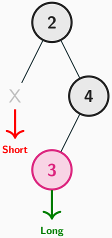
# Case 2: Deleting a BLACK Node (Uh oh...)

Delete **1** from our tree

### Before

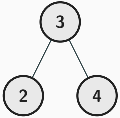



### After Delete




**✗ IMBALANCED**

### After Fix



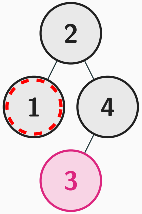

**✓ BALANCED**

- **Rotate:** Right at 4, then 2
- **Recolor:** 3 $\longrightarrow$ Black

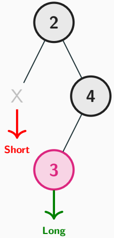
# Case 2: Deleting a BLACK Node (Uh oh...)

Delete **1** from our tree

**Before**

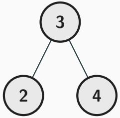



**After Delete**




✖ **IMBALANCED**

**After Fix**



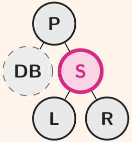

✔ **BALANCED**

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

# Black Node Deletion Cases 1 & 2

**P**=Parent **S**=Sibling **L/R**=S's children DB = Double-Black node

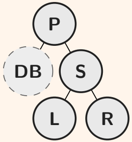
## ⇄ Case 1: Sibling is RED




**Fix:** Rotate P left, recolor S → Black, P → Red

*Converts to Case 2, 3, or 4*

## Case 2: Sibling & Children all BLACK




↑ **Fix:** Recolor S → Red,
push Double-Black up to P

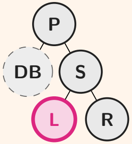
*Repeat fix from P if P was Black*

*Case 1 always leads to Case 2, 3, or 4 after rotation*

# Black Node Deletion Cases 3 & 4

**P**=Parent **S**=Sibling **L/R**=S's children DB = Double-Black node

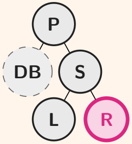
## Case 3: Sibling's Left child is RED




**Fix:** Right-rotate at S, swap colors of S & L

*Transforms into Case 4*

## Case 4: Sibling's Right child is RED

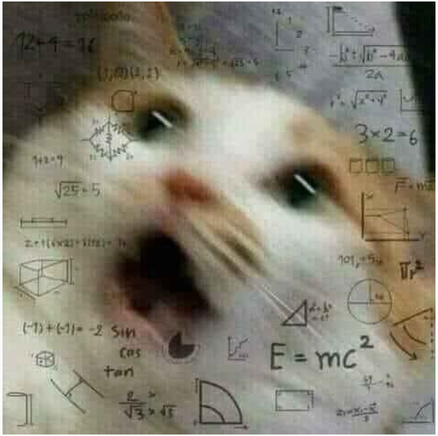



**Fix:** Left-rotate at P, recolor R $\rightarrow$ Black

**Double-Black resolved!**

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

# Umm... What?




## What just happened?

*"Four cases?*

*Rotations?*

*Recoloring?*

*Help!"*

Don't worry — even textbooks span 20+ pages on this.

# Don’t Worry!

**We know deletion is complex — and that’s *okay*!**

## The Reality

- The deletion algorithm is **huge** with many edge cases
- Each of the 4 cases has intricate implementation details
- Full implementation can span **hundreds** of lines

**We’ll skip the gory details for now!**

## If You’re Interested...

- CLRS Chapter 13 — full pseudocode & proofs
- Online visualizer: `visualgo.net`
- GitHub implementations in your favourite language

# Where Are Red-Black Trees Used?

**Everywhere!**

# Where Are Red-Black Trees Used?

**Everywhere!**

- **Linux Kernel**
  Process scheduling

# Where Are Red-Black Trees Used?

**Everywhere!**

- **Linux Kernel**
  Process scheduling
- **Java**
  TreeMap, TreeSet

# Where Are Red-Black Trees Used?

## Everywhere!

- **Linux Kernel**
  Process scheduling
- **Java**
  TreeMap, TreeSet
- **Databases**
  Indexing structures

# Where Are Red-Black Trees Used?

## Everywhere!

- **Linux Kernel**
  Process scheduling
- **Java**
  TreeMap, TreeSet
- **Databases**
  Indexing structures
- **File Systems**
  Directory organization

## Where Are Red-Black Trees Used?

**Everywhere!**

- **Linux Kernel**
  Process scheduling
- **Java**
  TreeMap, TreeSet
- **Databases**
  Indexing structures
- **File Systems**
  Directory organization

**Every time you use these...**

You're benefiting from **Red-Black Trees!**

# Red-Black Trees in a Nutshell

## Red-Black Trees in a Nutshell

- Complex but incredibly powerful

## Red-Black Trees in a Nutshell

- Complex but incredibly powerful
- Tricky to implement

## Red-Black Trees in a Nutshell

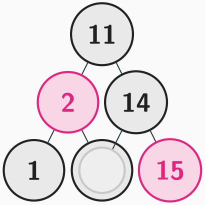
- Complex but incredibly powerful
- Tricky to implement
- Guaranteed $O(\log n)$ performance

## Red-Black Trees in a Nutshell

- Complex but incredibly powerful
- Tricky to implement
- Guaranteed $O(\log n)$ performance
- Used everywhere in computing

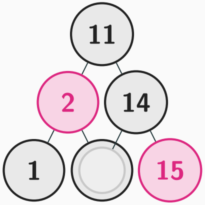
## Red-Black Trees in a Nutshell

- Complex but incredibly powerful
- Tricky to implement
- Guaranteed $O(\log n)$ performance
- Used everywhere in computing



## Red-Black Trees in a Nutshell

- Complex but incredibly powerful
- Tricky to implement
- Guaranteed $O(\log n)$ performance
- Used everywhere in computing



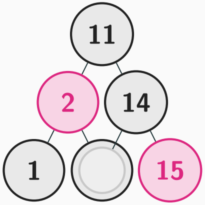

**Remember**

The next time you're struggling with RBT implementation...

Even the **inventor** moved on to Left-Leaning Red-Black Trees! ☻

# Red-Black Trees in a Nutshell

- Complex but incredibly powerful
- Tricky to implement
- Guaranteed $O(\log n)$ performance
- Used everywhere in computing




**Remember**

The next time you're struggling with RBT implementation...

Even the **inventor** moved on to Left-Leaning Red-Black Trees! 😃

# Any Questions?