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

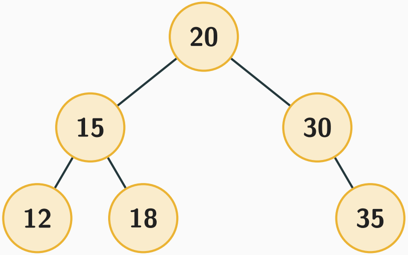
- Everything is **tree-structured**
- **Insert** data into the structure
- **Delete** data efficiently
- **Search** for data quickly

Good way to do all of this?

Use a BST!

The BST Rule

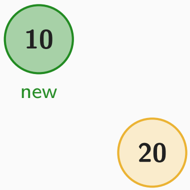
How does BST decide where to put a node?



The BST Rule

How does BST decide where to put a node?

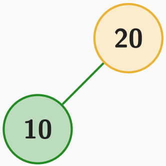
- **Smaller than me?** Go **Left**



The BST Rule

How does BST decide where to put a node?

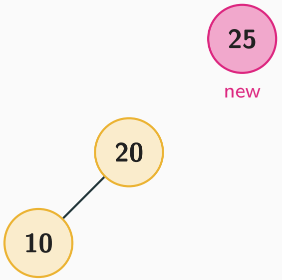
- **Smaller than me?** Go **Left**



The BST Rule

How does BST decide where to put a node?

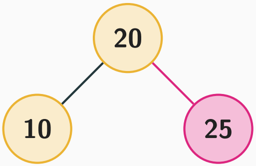
- **Smaller than me?** Go **Left**
- **Larger than me?** Go **Right**



The BST Rule

How does BST decide where to put a node?

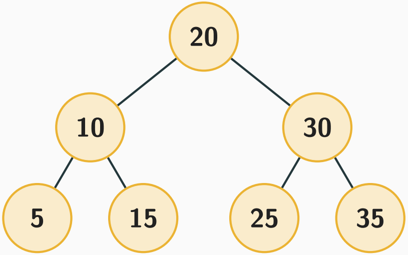
- **Smaller than me?** Go **Left**
- **Larger than me?** Go **Right**



The BST Rule

How does BST decide where to put a node?

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



Good technique!

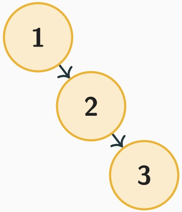
Insert the roll numbers in a class sequentially

1, 2, 3, 4 …10

Insert the roll numbers in a class sequentially

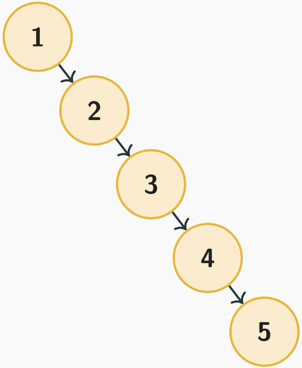
1, 2, 3, 4 ...10

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



Insert the roll numbers in a class sequentially

1, 2, 3, 4 ...10

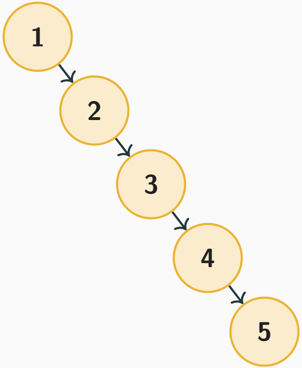


- Each goes to the **right** of the last
- The tree just keeps **growing** right...

...and on

Insert the roll numbers in a class sequentially

1, 2, 3, 4 ...10



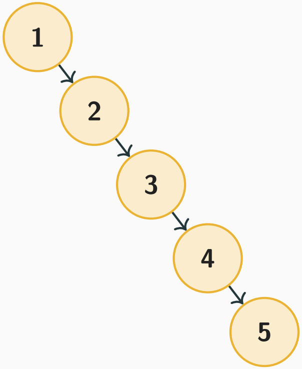
- Each goes to the **right** of the last
- The tree just keeps **growing** right...

Still works!

...and on

But, What's the Problem?

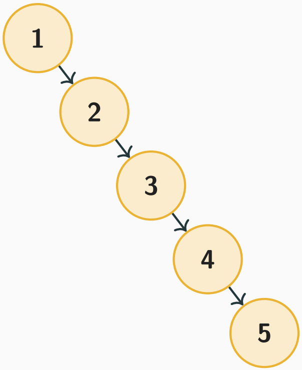
- **Height becomes n**



...and on

But, What's the Problem?

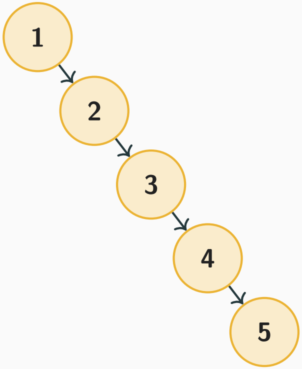
- **Height becomes n**
- Insertion takes $O(n)$



...and on

But, What's the Problem?

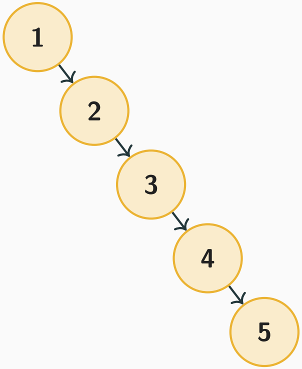
- **Height becomes n**
- Insertion takes $O(n)$
- Deletion takes $O(n)$



...and on

But, What's the Problem?

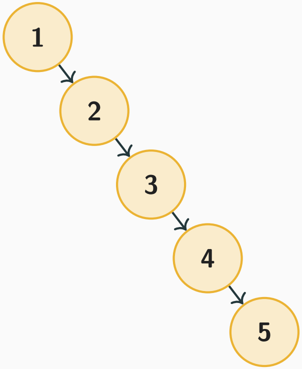
- **Height becomes n**
- Insertion takes $O(n)$
- Deletion takes $O(n)$
- Search takes $O(n)$



...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

The Solution?

Examples of Self-Balancing Trees:

- AVL Tree
- **Red-Black Tree**

The Solution?

Examples of Self-Balancing Trees:

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

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

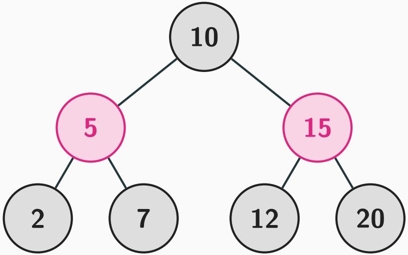
Hence, the name Red Black Tree

How does RBT do it:Properties

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

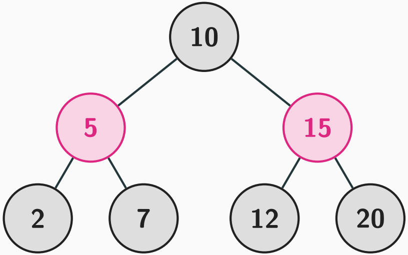
How does RBT do it:Properties

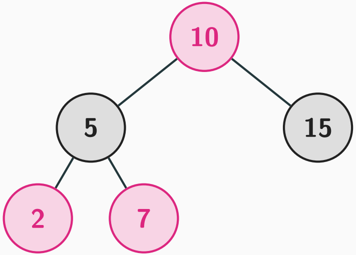
- **Property 2:** Root will always be a black node



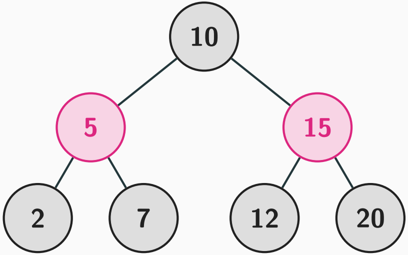
How does RBT do it:Properties

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

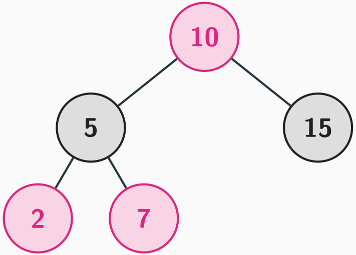




- **Property 2:** Root will always be a black node

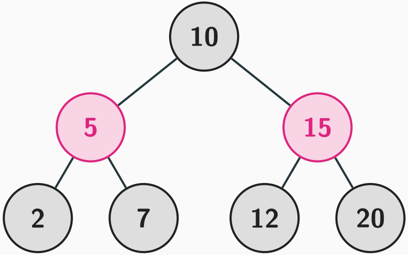


Correct

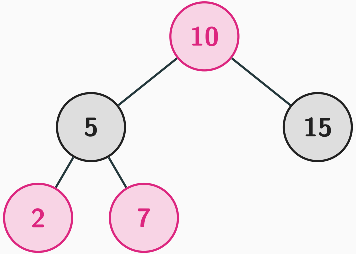


How does RBT do it:Properties

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



Correct



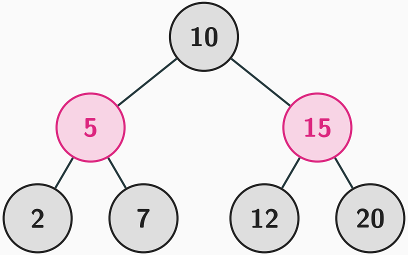
Incorrect

How does RBT do it:Properties

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

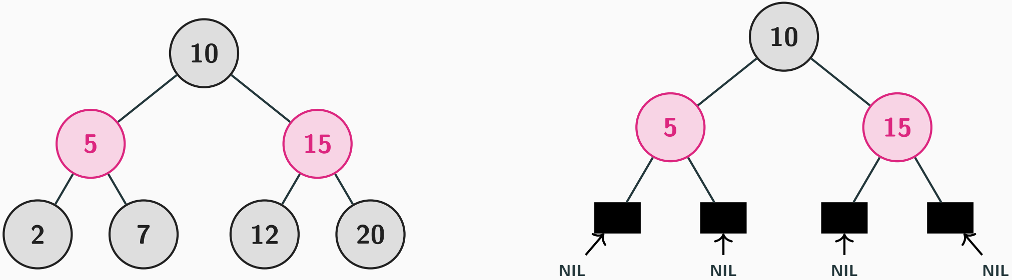
How does RBT do it:Properties

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



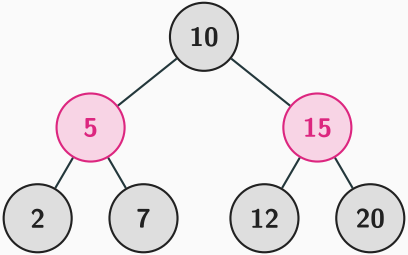
How does RBT do it:Properties

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

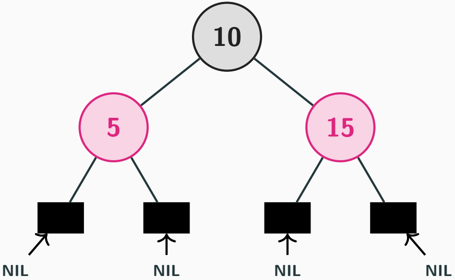


How does RBT do it:Properties

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

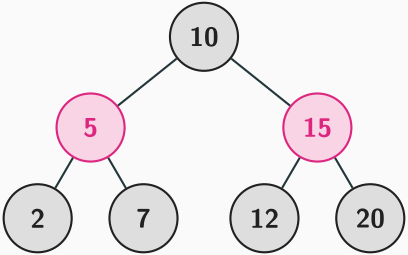


Black Leaves

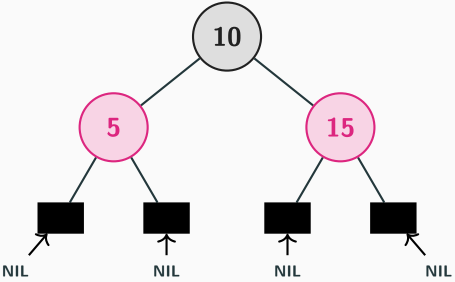


How does RBT do it:Properties

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



Black Leaves

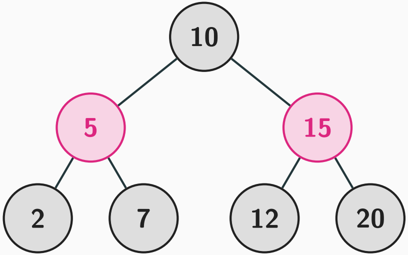


NIL nodes (counted as Black)

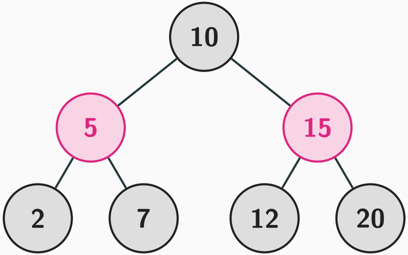
How does RBT do it:Properties

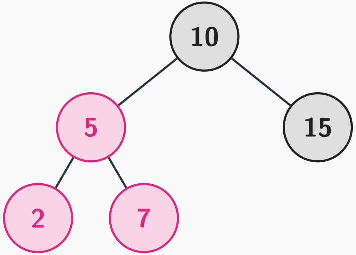
- **Property 4:** There will be no two consecutive red nodes

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

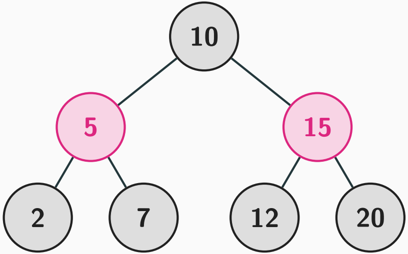


- **Property 4:** There will be no two consecutive red nodes

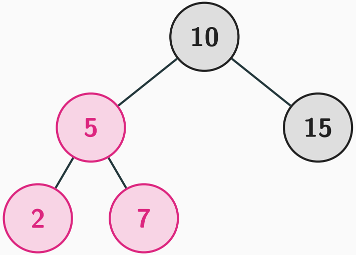




- **Property 4:** There will be no two consecutive red nodes

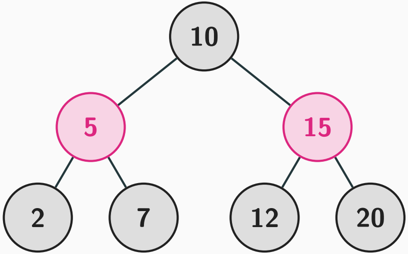


Correct

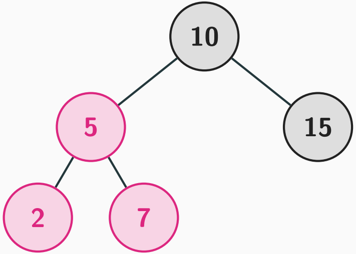


How does RBT do it:Properties

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



Correct



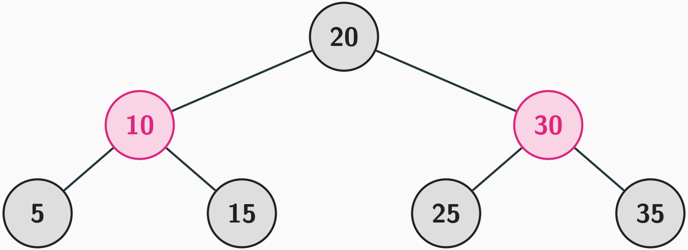
Incorrect

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

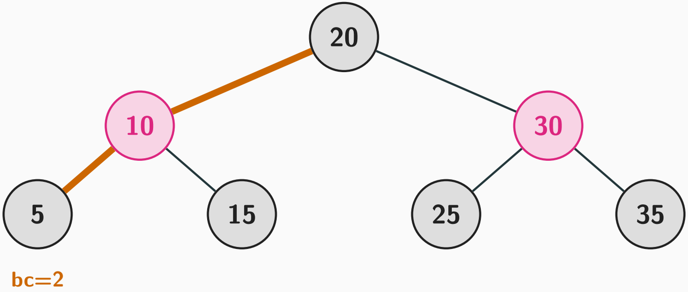
How does RBT do it:Properties

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



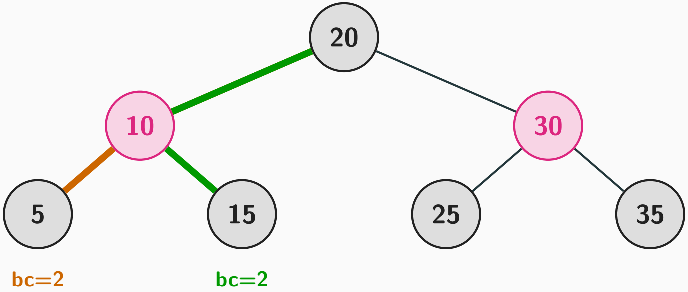
How does RBT do it:Properties

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



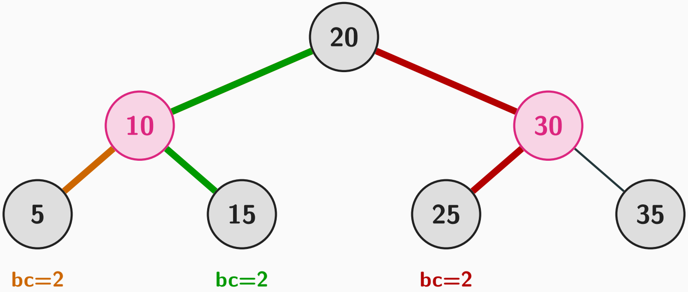
How does RBT do it:Properties

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



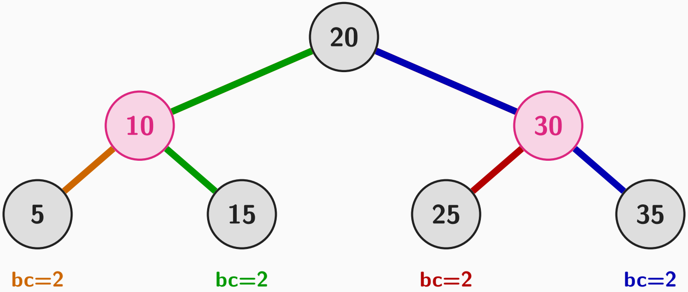
How does RBT do it:Properties

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



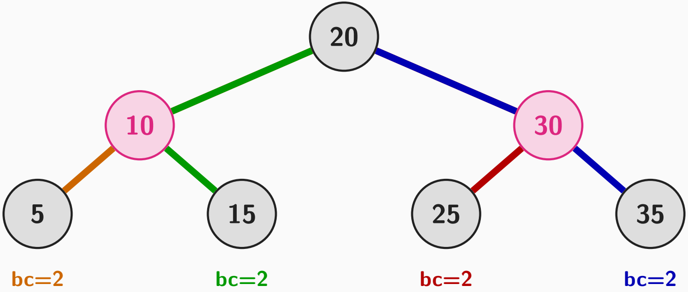
How does RBT do it:Properties

- **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?

RBT Operations

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

Let’s see some operations....

Insertion

Insertion

Insert node x in a Red Black Tree

Insertion

Insert node x in a Red Black Tree

Pseudocode

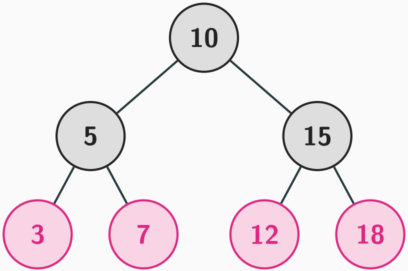
```
color[x] = RED

y = root[T]

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

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]
```

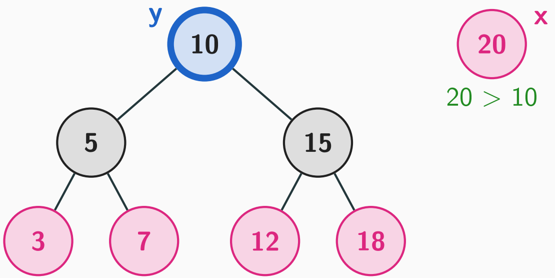
Existing RBT



Insert node x in a Red Black Tree

Pseudocode

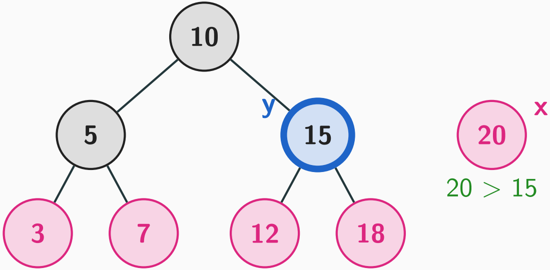
```
color[x] = RED

y = root[T]

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



Insert node x in a Red Black Tree

Pseudocode

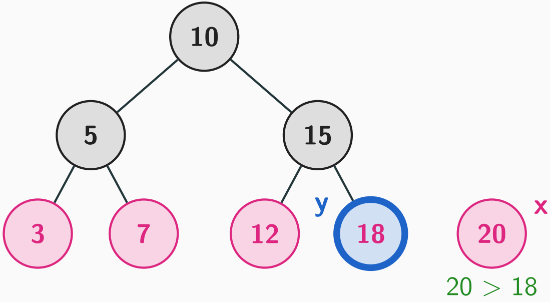
```
color[x] = RED

y = root[T]

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



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]
```

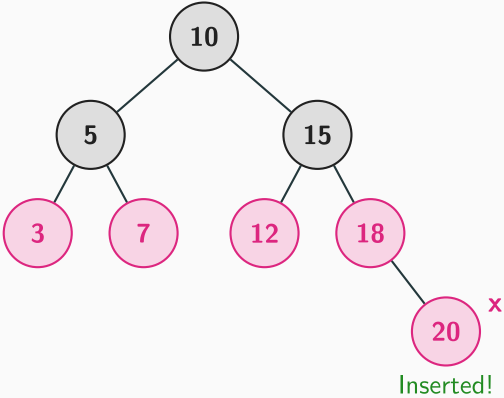


Insertion

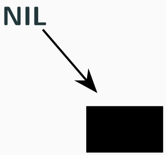
Insert node x in a Red Black Tree

Pseudocode

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

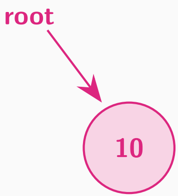


Insert 10



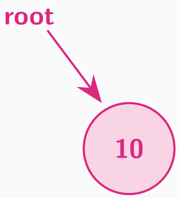
What can go wrong

Insert 10



What can go wrong

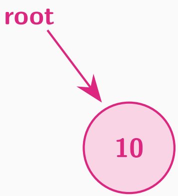
Insert 10



Root can't be RED

What can go wrong

Insert 10

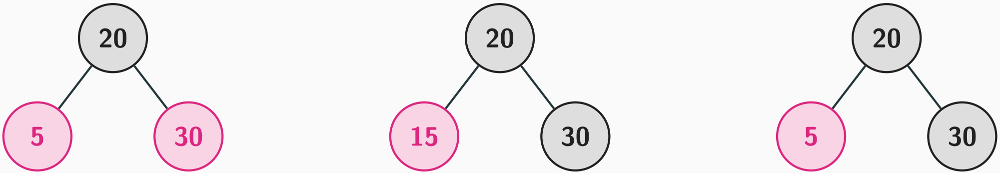


Root can't be RED

Case 1

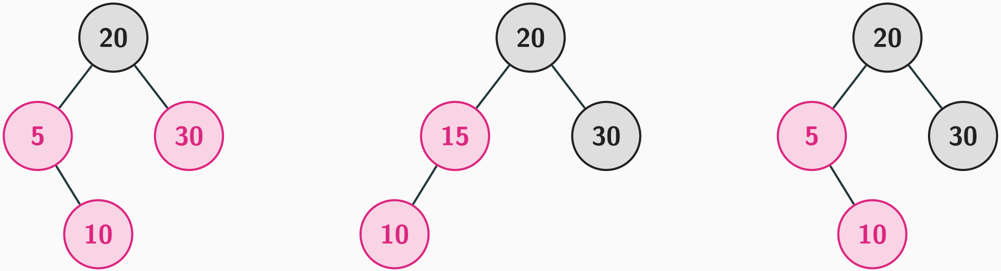
What can go wrong

Insert 10



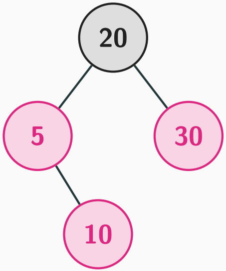
What can go wrong

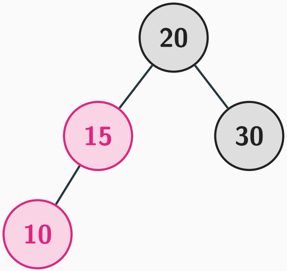
Insert 10

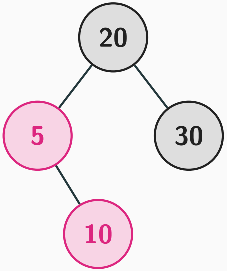


What can go wrong

Insert 10







Case 2 **Case 3** **Case 4**

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

	Violation	Fix
Case 1	Root is **RED** — Property 2 broken	Recolor root to **BLACK**
Case 2	Uncle is **RED** — two reds adjacent	Recolor parent, uncle **BLACK**; grandparent **RED**
Case 3	Uncle is **BLACK** — Right-Right	Left rotate at grandparent, then recolor

Solutions to Violations

	Violation	Fix
Case 1	Root is **RED** — Property 2 broken	Recolor root to **BLACK**
Case 2	Uncle is **RED** — two reds adjacent	Recolor parent, uncle **BLACK**; grandparent **RED**
Case 3	Uncle is **BLACK** — Right-Right	Left rotate at grandparent, then recolor
Case 4	Uncle is **BLACK** — Left-Right	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

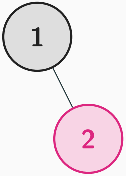
Insert 2

- 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** ✔

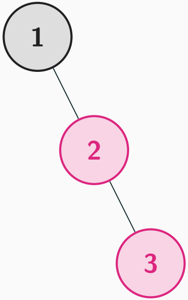
After Insert



Insert 3

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

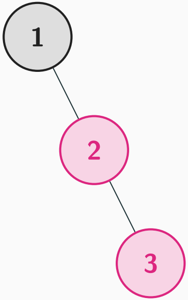
After Insert



Insert 3

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

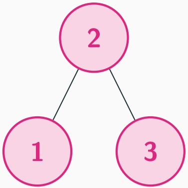
Violation Found — Two RED in a row!



Insert 3

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

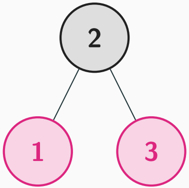
After Left Rotation



Insert 3

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

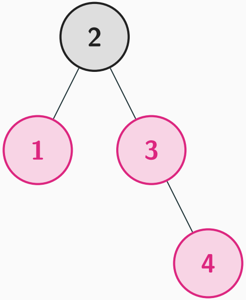
After Recolor



Insert 4

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

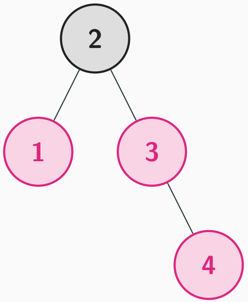
After Insert



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



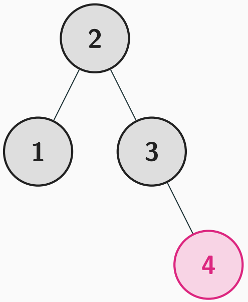
Insert 4

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

After Recolor



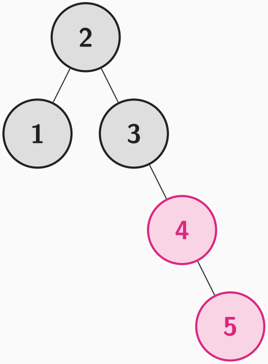
Insert 5

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

Insert 5

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

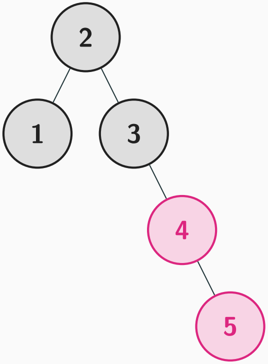
After Insert



Insert 5

- Right child of 4
- Insert as **RED** (default)
- Uncle (node 1) is **BLACK**
- **CASE 3**
  Uncle is **BLACK** — rotate & recolor!

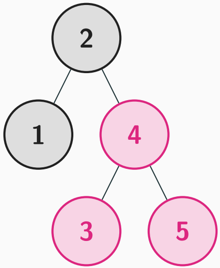
Violation — Two RED in a row!



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 $\rightarrow$ 4 moves up

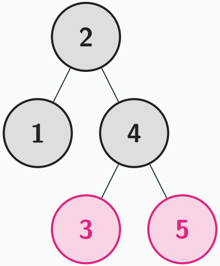
After Left Rotation



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!**

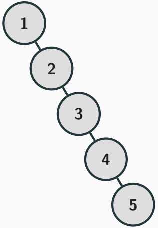
After Recolor



BST vs. Red-Black Tree

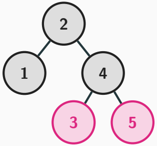
Inserting $\{1, 2, 3, 4, 5\}$ in order

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

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

- No problem!

Deleting a BLACK node

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

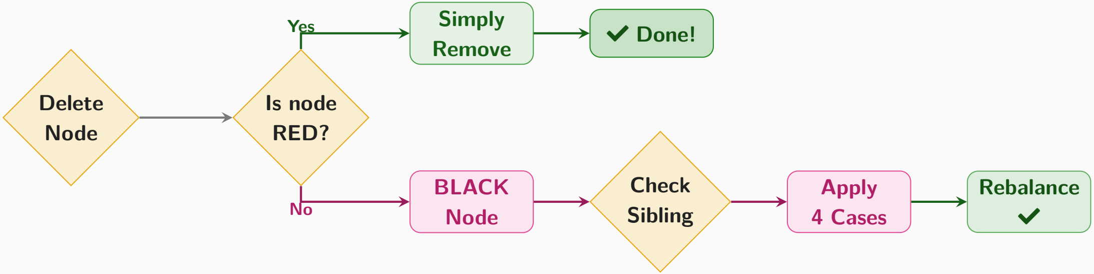
- No problem!
- Just remove it
- **Properties still hold**

Deleting a BLACK node

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

Let's see both cases...

Deletion Decision Flowchart



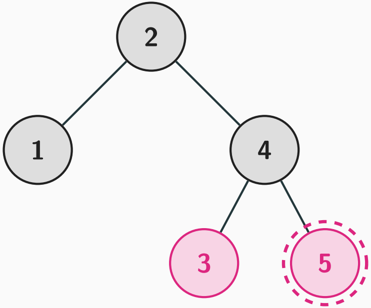
Top path (RED node) = straightforward

Bottom path (BLACK node) = complex

Case 1: Deleting a RED Node

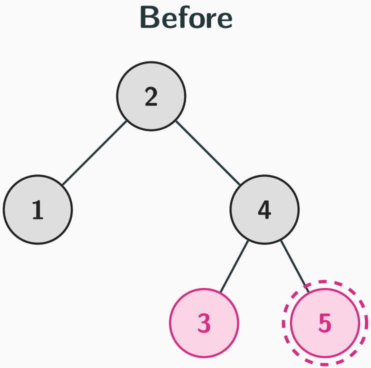
Delete node **5** from the tree

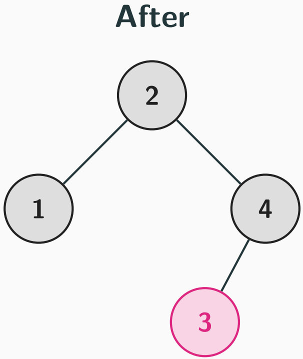
Before



Case 1: Deleting a RED Node

Delete node **5** from the tree





Case 1: Deleting a RED Node

Is it done?

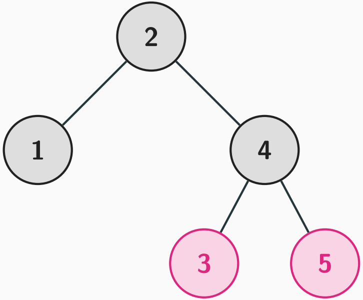
Is it done?

Let's check the black height.

Case 1: Black-Height Stays the Same

Every path still has **bc = 2** black nodes after removing 5

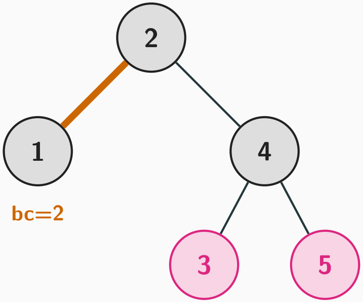
Before (with node 5)



Case 1: Black-Height Stays the Same

Every path still has **bc = 2** black nodes after removing 5

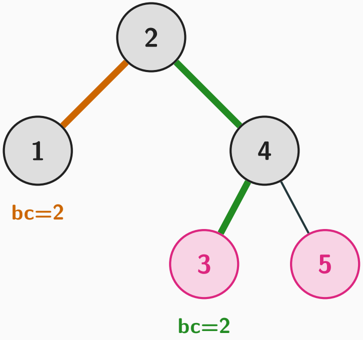
Before (with node 5)



Case 1: Black-Height Stays the Same

Every path still has **bc = 2** black nodes after removing 5

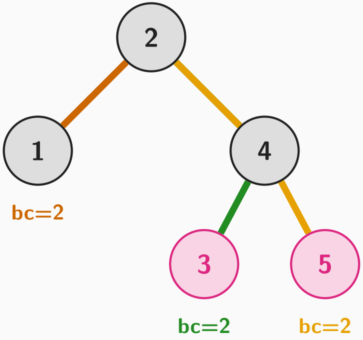
Before (with node 5)



Case 1: Black-Height Stays the Same

Every path still has **bc = 2** black nodes after removing 5

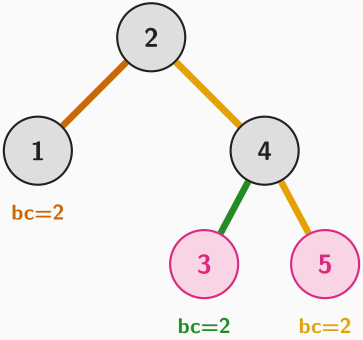
Before (with node 5)



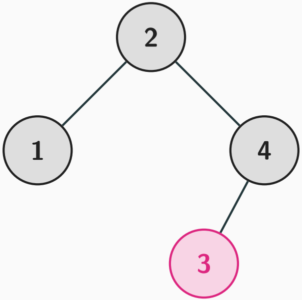
Case 1: Black-Height Stays the Same

Every path still has **bc = 2** black nodes after removing 5

Before (with node 5)



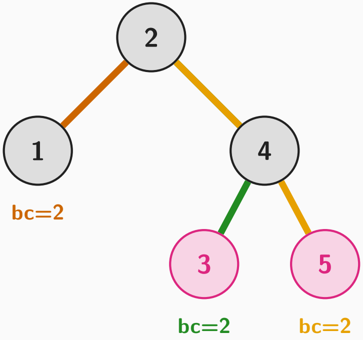
After (node 5 removed)



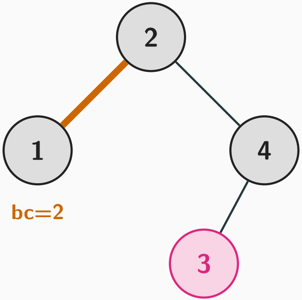
Case 1: Black-Height Stays the Same

Every path still has **bc = 2** black nodes after removing 5

Before (with node 5)

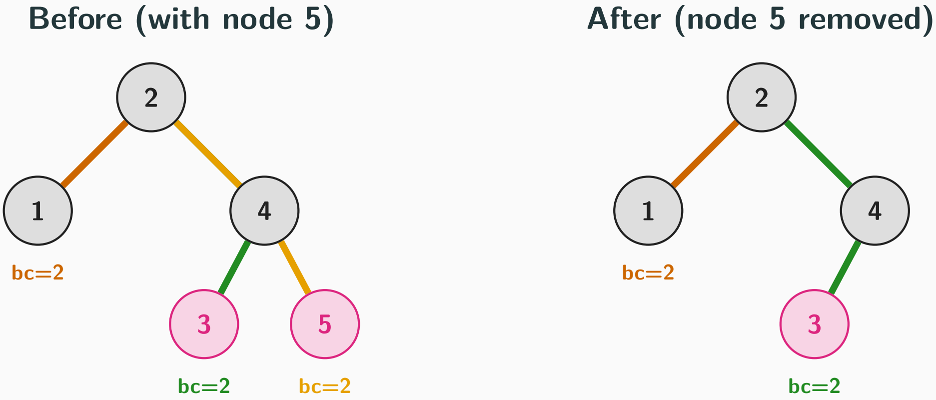


After (node 5 removed)



Case 1: Black-Height Stays the Same

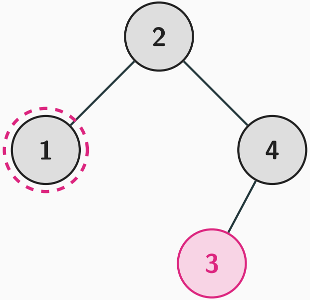
Every path still has $\mathbf{bc = 2}$ black nodes after removing 5



Case 2: Deleting a BLACK Node

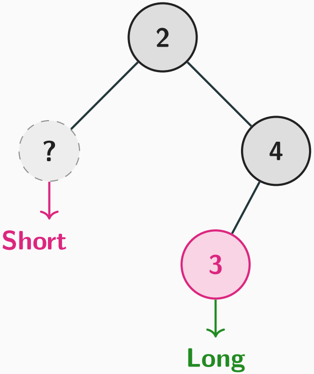
Delete node **1** from the tree

Before deletion:



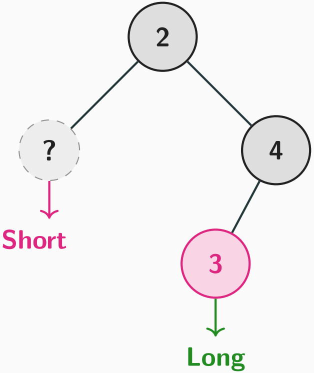
Case 2: After Deleting the BLACK Node

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



Case 2: After Deleting the BLACK Node

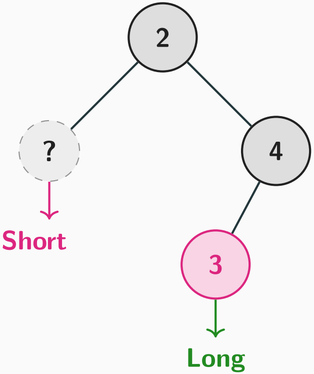
The node is gone - but now we have a **problem**



- Left path is now **shorter**

Case 2: After Deleting the BLACK Node

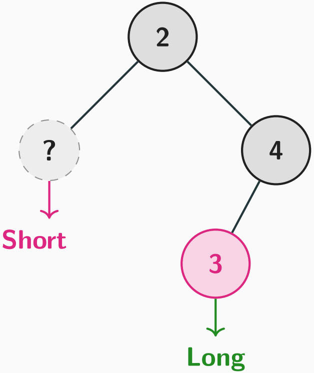
The node is gone - but now we have a **problem**



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

Case 2: After Deleting the BLACK Node

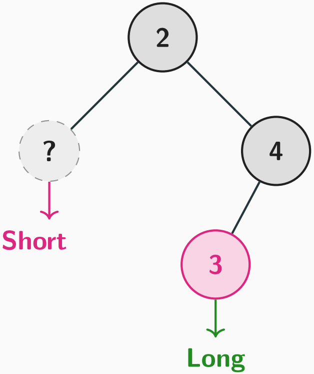
The node is gone - but now we have a **problem**



- 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

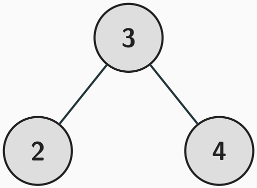
- **Rotate:** Right at 4, then left at 2

Case 2: The Fix-up

- **Rotate:** Right at 4, then left at 2
- **Recolor:** Node 3 $\rightarrow$ Black

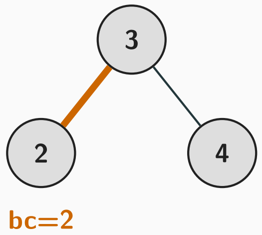
Case 2: The Fix-up

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



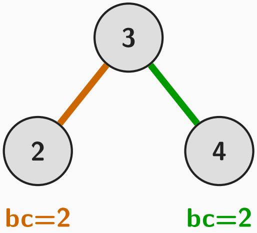
Case 2: The Fix-up

- **Rotate:** Right at 4, then left at 2
- **Recolor:** Node 3 $\rightarrow$ Black

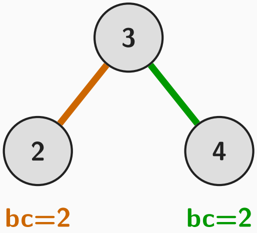


Case 2: The Fix-up

- **Rotate:** Right at 4,
  then left at 2
- **Recolor:** Node 3 $\rightarrow$ Black



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



All paths: $\mathbf{bc = 2}$ - 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

DB = 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**.

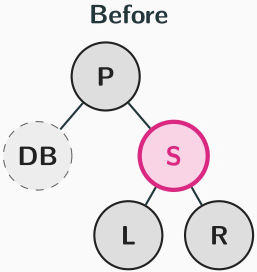
P = Parent **S** = Sibling **L** / **R** = S's children

DB = Double-Black node

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

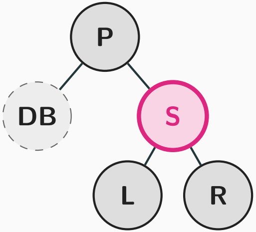
Fix Case 1 of 4: Sibling is RED

The Sibling S is RED



The Sibling S is RED

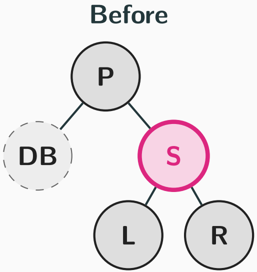
Before



- **Rotate** P to the left

Fix Case 1 of 4: Sibling is RED

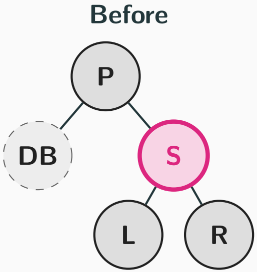
The Sibling S is RED



- **Rotate** P to the left
- **Recolor:** S $\rightarrow$ Black, P $\rightarrow$ Red

Fix Case 1 of 4: Sibling is RED

The Sibling S is RED



- **Rotate** P to the left
- **Recolor:** S $\rightarrow$ Black, P $\rightarrow$ Red

Fix Case 1 of 4: Sibling is RED

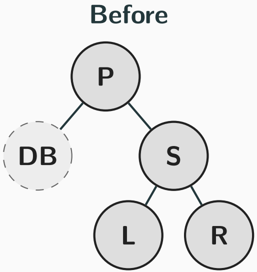
The Sibling S is RED

Now apply Case 2, 3, or 4 to DB

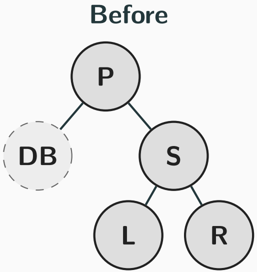
- **Rotate** P to the left
- **Recolor:** S $\rightarrow$ Black, P $\rightarrow$ Red

Fix Case 2 of 4: Sibling & Children All BLACK

S and both children are BLACK



S and both children are BLACK

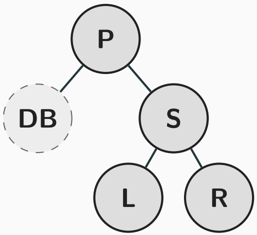


- **Recolor** S $\rightarrow$ Red

Fix Case 2 of 4: Sibling & Children All BLACK

S and both children are BLACK

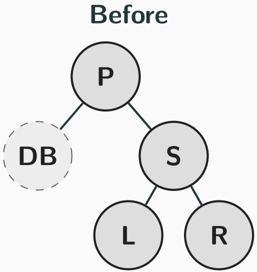
Before



- **Recolor** S $\rightarrow$ 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 $\rightarrow$ Red
- Push the Double-Black **up to P**

Fix Case 2 of 4: Sibling & Children All BLACK

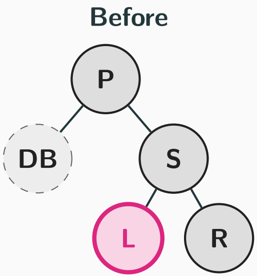
S and both children are BLACK

DB pushed to P — continue fixing

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

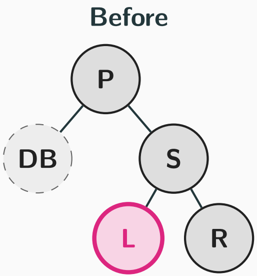
Fix Case 3 of 4: Sibling's Left Child is RED

S is Black, S's Left child is RED



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

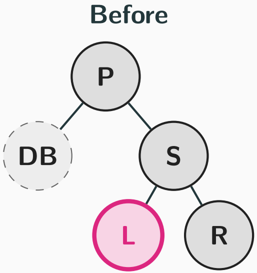
S is Black, S's Left child is RED



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

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

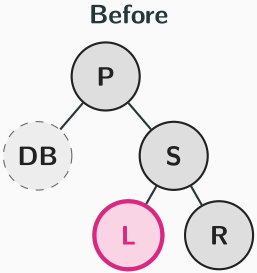
S is Black, S's Left child is RED



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

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

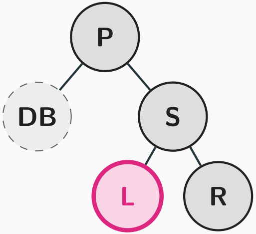
S is Black, S's Left child is RED

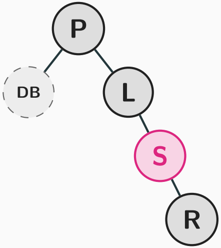


- **Right-rotate** at S, & **Swap colors** of S and L

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

S is Black, S's Left child is RED



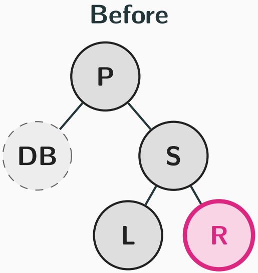


Now proceed with Case 4

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

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

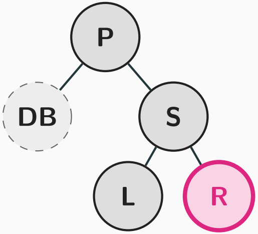
S is Black, S's Right child is RED



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

S is Black, S's Right child is RED

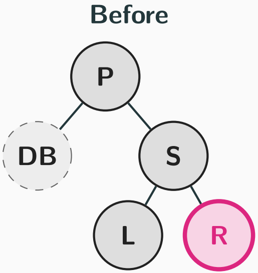
Before



- **Left-rotate** at P

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

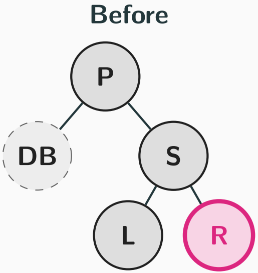
S is Black, S's Right child is RED



- **Left-rotate** at P
- **Recolor** R $\rightarrow$ Black

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

S is Black, S's Right child is RED

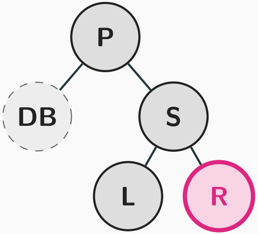


- **Left-rotate** at P
- **Recolor** R → Black

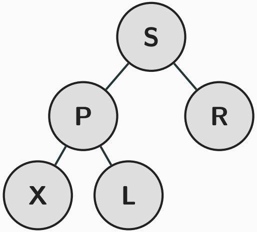
Fix Case 4 of 4: Sibling's Right Child is RED

S is Black, S's Right child is RED

Before



After



Double-Black fully resolved!

- **Left-rotate** at P
- **Recolor** R $\rightarrow$ Black

Summary

The cases form a **chain**:

Summary

The cases form a **chain**:

$$\textbf{Case 1} \xrightarrow{\text{rotate}} \textbf{Case 2/3} \xrightarrow{\text{transform}} \textbf{Case 4} \xrightarrow{\text{rotate}} \checkmark\ \textbf{Done!}$$

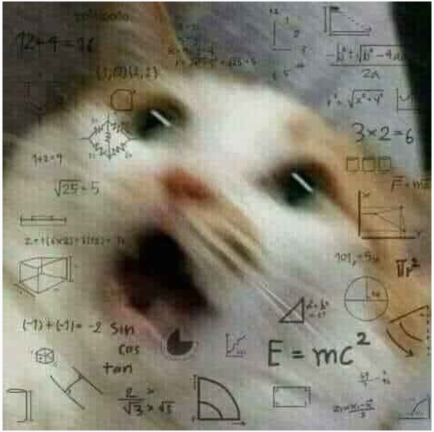
Goal: Eventually reach Case 4 to fully eliminate Double-Black

Summary

The cases form a **chain**:

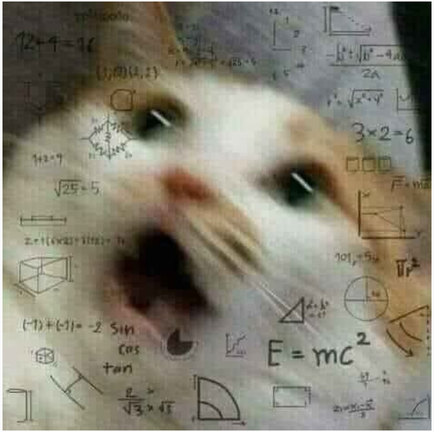
Goal: Eventually reach Case 4 to fully eliminate Double-Black

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



Confused?

Too Many Cases?



Confused?

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

Even seasoned programmers keep a reference sheet open for this one.

The Reality of Deletion

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

The Reality of Deletion

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

- The deletion algorithm is **huge** with many edge cases

The Reality of Deletion

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

- The deletion algorithm is **huge** with many edge cases
- Each of the 4 cases has intricate implementation details

The Reality of Deletion

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

- 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 of code

The Reality of Deletion

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

- 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 of code

We'll skip the gory details for now!

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.

Rotation Complexity: $O(1)$ Operations

Rotation Complexity: $O(1)$ Operations

Left Rotation:

- Restructures tree locally

Right Rotation:

- Mirror of left rotation

Why Rotations Work: The Magic Behind Balance

Rotation Complexity: $O(1)$ Operations

Left Rotation:

- Restructures tree locally
- Preserves binary search property

Right Rotation:

- Mirror of left rotation
- Same time complexity

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

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 modify a **constant number of pointers**!

No tree traversal required.

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

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

Key Insight

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

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

Key Insight

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

Proof Sketch:

1. Every path has **same black-height**
2. **No red-red** children
3. $\geq 50\%$ nodes on path are **black**

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

Key Insight

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

Proof Sketch:

1. Every path has **same black-height**
2. **No red-red** children
3. $\geq 50\%$ nodes on path are **black**
4. Height $\leq 2\times$ black-height

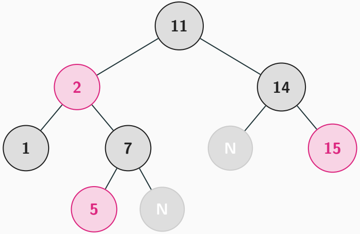
Height Proof: Why $O(\log n)$?

Key Insight

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

Proof Sketch:

1. Every path has **same black-height**
2. **No red-red** children
3. $\geq 50\%$ nodes on 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

Why Red-Black Trees Are Space Efficient

Space Complexity: $O(n)$ Memory Usage

Memory Requirements:

- Each node stores: **key + color + 2 pointers**

Comparison:

- AVL trees: height field per node

Space Complexity: $O(n)$ Memory Usage

Memory Requirements:

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

Comparison:

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

Why Red-Black Trees Are Space Efficient

Space Complexity: $O(n)$ Memory Usage

Memory Requirements:

- Each node stores: **key + color + 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

Why Red-Black Trees Are Space Efficient

Space Complexity: $O(n)$ Memory Usage

Memory Requirements:

- Each node stores: **key + color + 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

Key Insight

RBTs achieve $O(n)$ **space** with **1 bit** per node (color field).

The Evolution: From Red-Black to Modern Trees

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**]

The Evolution: From Red-Black to Modern Trees

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?

"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?

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

Note: Even Sedgewick evolved the design—Left-Leaning RBTs reduce implementation complexity.

Red-Black Trees in Modern Tech

Red-Black Trees in Modern Tech

Traditional Uses:

- **Scheduling**: CPU, process queues

Red-Black Trees in Modern Tech

Traditional Uses:

- **Scheduling**: CPU, process queues
- **Memory**: Allocation, defragmentation

Red-Black Trees in Modern Tech

Traditional Uses:

- **Scheduling**: CPU, process queues
- **Memory**: Allocation, defragmentation
- **Routing**: Network tables, prefixes

Red-Black Trees in Modern Tech

Traditional Uses:

- **Scheduling**: CPU, process queues
- **Memory**: Allocation, defragmentation
- **Routing**: Network tables, prefixes
- **Graphics**: Spatial indexing, trees

Red-Black Trees in Modern Tech

Traditional Uses:

- **Scheduling**: CPU, process queues
- **Memory**: Allocation, defragmentation
- **Routing**: Network tables, prefixes
- **Graphics**: Spatial indexing, trees

Modern Applications:

Red-Black Trees in Modern Tech

Traditional Uses:

- **Scheduling**: CPU, process queues
- **Memory**: Allocation, defragmentation
- **Routing**: Network tables, prefixes
- **Graphics**: Spatial indexing, trees

Modern Applications:

- **ML indexing**, **cloud storage**, **blockchain pathfinding**

Red-Black Trees in Modern Tech

Traditional Uses:

- **Scheduling**: CPU, process queues
- **Memory**: Allocation, defragmentation
- **Routing**: Network tables, prefixes
- **Graphics**: Spatial indexing, trees

Modern Applications:

- **ML indexing**, **cloud storage**, **blockchain pathfinding**
- Trusted by billions of devices daily

Why Choose Red-Black 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	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

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

- **Fewer rotations** than AVL (faster inserts)
- **Simpler code** than B-trees

The Trade-off

Balanced performance across all operations—optimal for general-purpose use.

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

- **Fewer rotations** than AVL (faster inserts)
- **Simpler code** than B-trees
- **Guaranteed** $O(\log n)$ all operations

The Trade-off

Balanced performance across all operations—optimal for general-purpose use.

Industry Secrets: What Big Tech Uses

The Real Story Behind the Scenes

The Real Story Behind the Scenes

Tech Giants & RBTs:

- **Google**: MapReduce distributed sorting

The Real Story Behind the Scenes

Tech Giants & RBTs:

- **Google**: MapReduce distributed sorting
- **Facebook**: Feed ranking systems

The Real Story Behind the Scenes

Tech Giants & RBTs:

- **Google**: MapReduce distributed sorting
- **Facebook**: Feed ranking systems
- **Amazon**: Catalog indexing

The Real Story Behind the Scenes

Tech Giants & RBTs:

- **Google**: MapReduce distributed sorting
- **Facebook**: Feed ranking systems
- **Amazon**: Catalog indexing
- **Netflix**: Content routing

Industry Secrets: What Big Tech Uses

The Real Story Behind the Scenes

Tech Giants & RBTs:

- **Google**: MapReduce distributed sorting
- **Facebook**: Feed ranking systems
- **Amazon**: Catalog indexing
- **Netflix**: Content routing
- **Linux**: Process scheduling

The Real Story Behind the Scenes

Tech Giants & RBTs:

- **Google**: MapReduce distributed sorting
- **Facebook**: Feed ranking systems
- **Amazon**: Catalog indexing
- **Netflix**: Content routing
- **Linux**: Process scheduling

Production Strategy

Hybrid approaches: RBTs for small-medium datasets, B-trees for disk-based storage.

Industry Secrets: What Big Tech Uses

The Real Story Behind the Scenes

Tech Giants & RBTs:

- **Google**: MapReduce distributed sorting
- **Facebook**: Feed ranking systems
- **Amazon**: Catalog indexing
- **Netflix**: Content routing
- **Linux**: Process scheduling

Production Strategy

Hybrid approaches: RBTs for small-medium datasets, B-trees for disk-based storage.

Next Generation Data Structures

Next Generation Data Structures

Emerging Trends:

- **Persistent Trees**: Functional
  correctness guarantees

Next Generation Data Structures

Emerging Trends:

- **Persistent Trees**: Functional correctness guarantees
- **Lock-free RBTs**: Concurrent access patterns

Next Generation Data Structures

Emerging Trends:

- **Persistent Trees**: Functional correctness guarantees
- **Lock-free RBTs**: Concurrent access patterns
- **GPU data structures**: Massive parallelization

Next Generation Data Structures

Emerging Trends:

- **Persistent Trees**: Functional correctness guarantees
- **Lock-free RBTs**: Concurrent access patterns
- **GPU data structures**: Massive parallelization
- **Learned indices**: ML-augmented trees

Next Generation Data Structures

Emerging Trends:

- **Persistent Trees**: Functional correctness guarantees
- **Lock-free RBTs**: Concurrent access patterns
- **GPU data structures**: Massive parallelization
- **Learned indices**: ML-augmented trees

Next Frontier

Adaptive structures: learning access patterns to optimize tree shape dynamically.

Why Red-Black Trees Matter

Why Red-Black Trees Matter

- $O(\log n)$ **guaranteed**

Why Red-Black Trees Matter

- $O(\log n)$ **guaranteed**
- **Practical efficiency**—fewer rotations

Why Red-Black Trees Matter

- $O(\log n)$ **guaranteed**
- **Practical efficiency**—fewer rotations
- **Mission-critical systems**—everywhere

Why Red-Black Trees Matter

- $O(\log n)$ **guaranteed**
- **Practical efficiency**—fewer rotations
- **Mission-critical systems**—everywhere

The Foundation

35+ years of proven reliability in production systems.

Why Red-Black Trees Matter

- $O(\log n)$ **guaranteed**
- **Practical efficiency**—fewer rotations
- **Mission-critical systems**—everywhere

The Foundation

35+ years of proven reliability in production systems.

Words to Live By

"We know exactly how to balance a tree.
We just haven't been outside to see one."

— Unknown

Thanks for listening!