ECE2810J

Data Structures and Algorithms

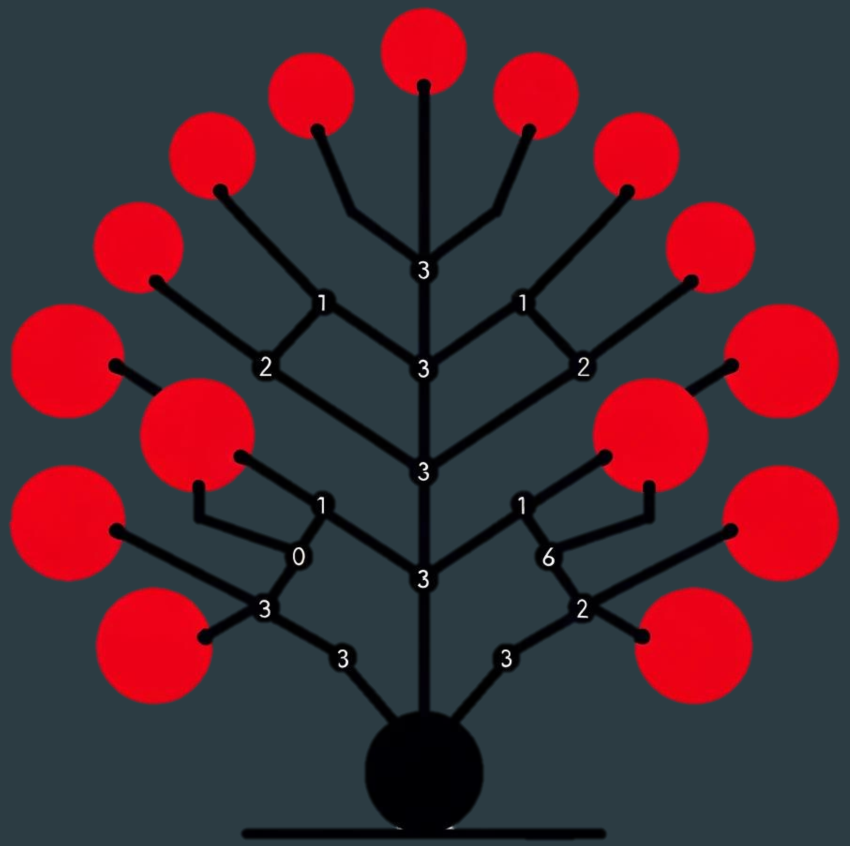
Red-black Trees

Learning Objectives:

- Know what a red-black tree is and its properties
- Know how to do insertion for a red-black tree

Outline

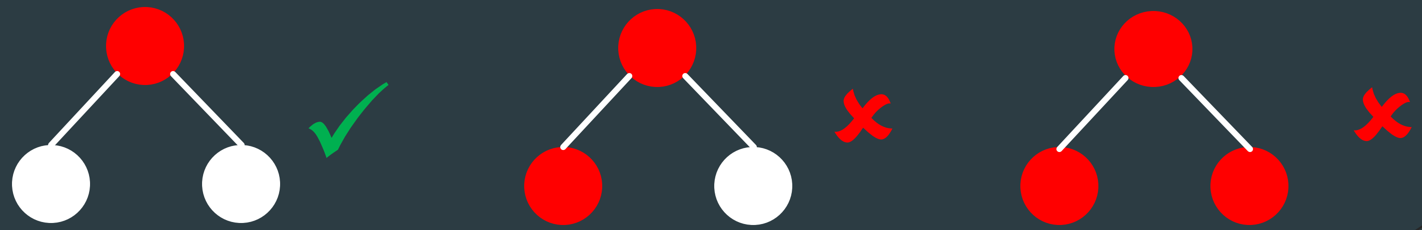
- Red-black Trees: Basics
- Red-black Trees: Insertion



Red-Black Tree

- A binary search tree. The data structure requires an extra one-bit color field in each node.
- Property

1. Every node is either red or black (we use white for better visualization).
2. **Root rule**: The root is black.
3. **Red rule**: Red node can only have black children.
   - Can't have two consecutive red nodes on a path.

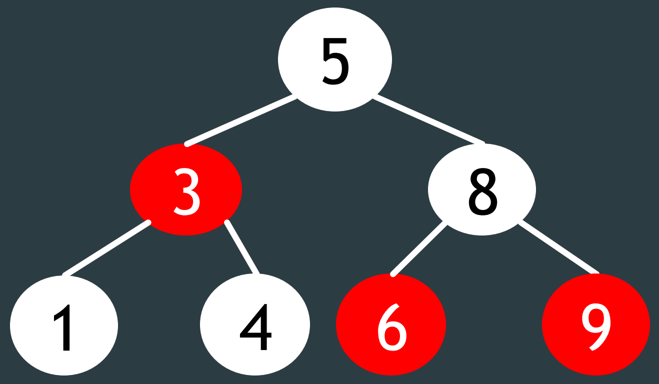


4. **Path rule**: Every path from a node x to NULL must have the same number of black nodes (including x itself).

Red-Black Tree Example

- Property

1. A binary search tree
2. Every node is either red or black (we use white for better visualization).
3. **Root rule**: The root is black.
4. **Red rule**: Red node can only have black children.
5. **Path rule**: Every path from a node x to NULL must have the same number of black nodes (including x itself).

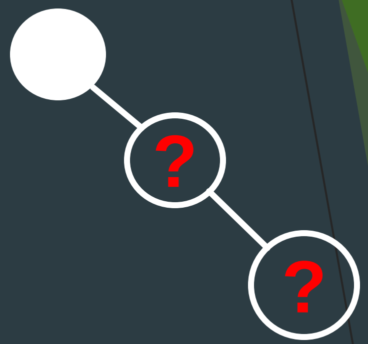


Counter Example

- Property

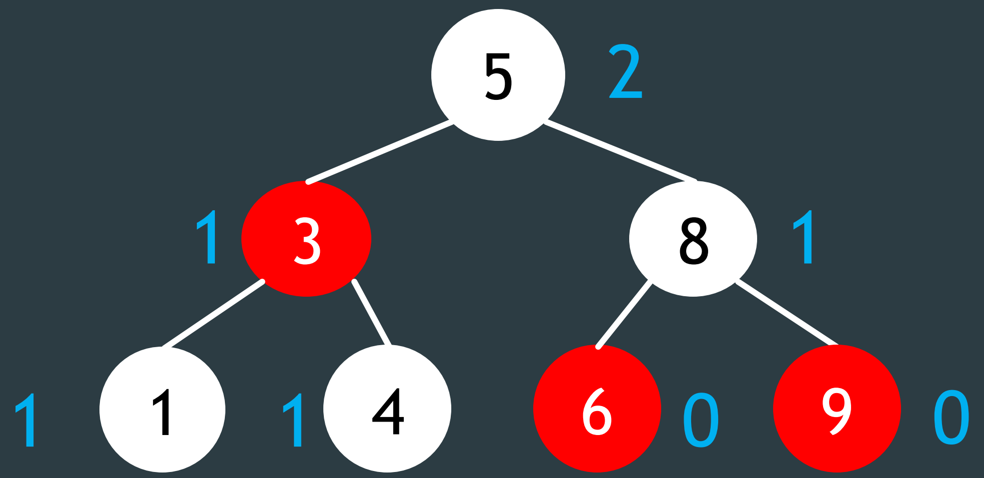
1. A binary search tree
2. Every node is either red or black.
3. **Root rule**: The root is black.
4. **Red rule**: Red node can only have black children.
5. **Path rule**: Every path from a node x to NULL must have the same number of black nodes (including x itself).

- <u>Claim</u>: a chain of length 3 cannot be a red-black tree



Black Height

- Black height of a node x is the number of black nodes on the path from x to NULL, including x itself.



Which Statements Are Correct?

A. It is possible for a red node to have a single child.

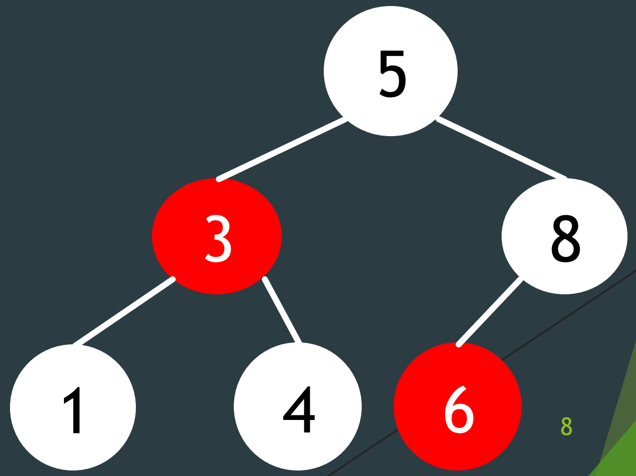
B. It is possible for a **black** node to have a single child.

C. It is possible for a node to have two children of different colors.

D. It is possible for a node to have two children and the node and its children are all of the same color.

Implication of the Rules

- If a red node has at least one child, it **must have** two children and they must be black.
  - Why?
    - A red node's child can only be black.
    - If has only one black child, then violate the path rule.
- If a black node has only one child, that child **must be** a red leaf.
  - Why?
    - Can't be black.
    - Must be a leaf.



Height Guarantee

- **Claim**: every red-black tree with n nodes has height $\leq 2\log_2(n+1)$.
- Proof:
  - In a binary tree with n nodes, there is a root-NULL path with **at most** $\log_2(n+1)$ nodes. (why?)
    - **Thus**: # black nodes on that path $\leq \log_2(n+1)$.
  - By **path rule**: every root-NULL path has $\leq \log_2(n+1)$ black nodes.
  - By **red rule**: every root-NULL path has $\leq 2\log_2(n+1)$ total nodes.

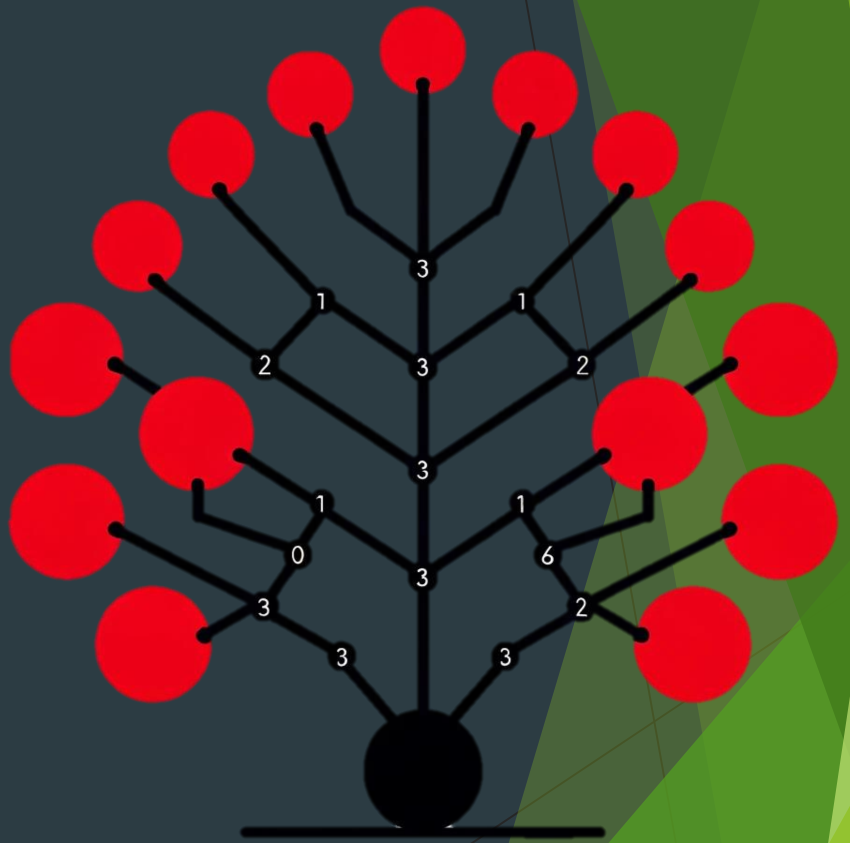
Operations on Red-Black Trees

- All query operations (e.g., search, min, max, succ, pred) work just like those on general BST.
  - They run in $O(\log n)$ time on a red-black trees with n nodes in the worst case.

- The modifying operations "insertion" and "removal" must maintain the red-black tree properties.
  - They are complex.

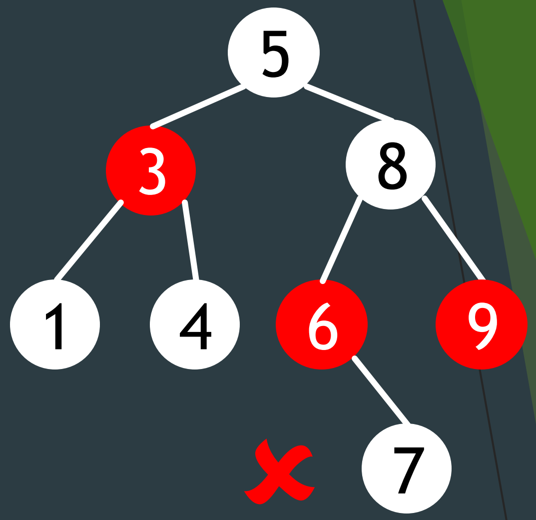
Outline

- Red-black Trees: Basics
- Red-black Trees: Insertion

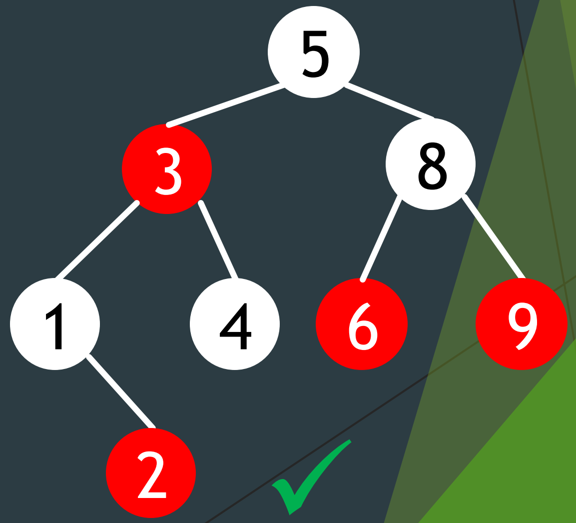


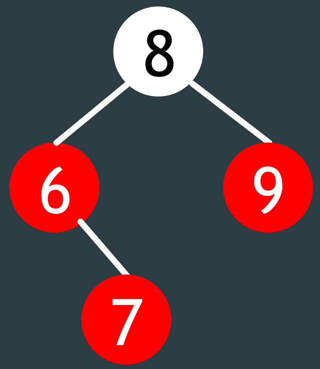
Insertion

- New node is always a leaf.
  - However, it can't be **black**!
    - Otherwise, violate path rule.
  - Therefore the new leaf must be red.



- If parent is black, done (trivial case).
- If parent is red, violate the red rule!

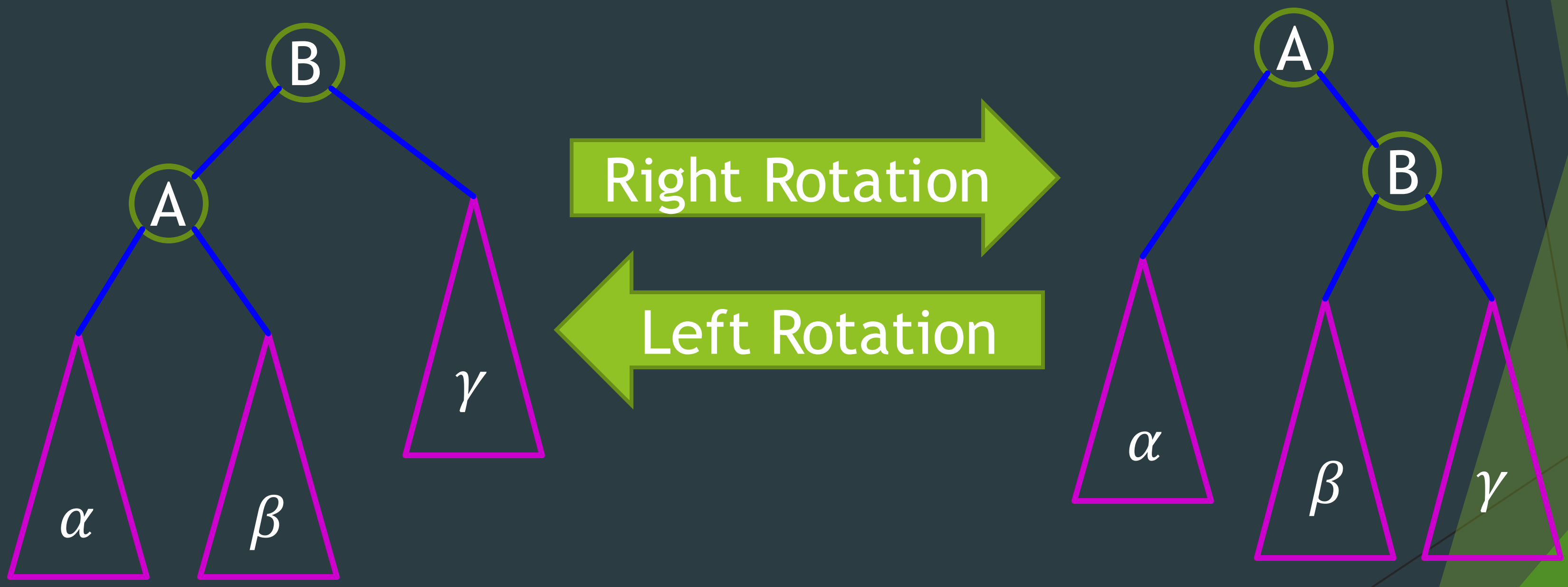




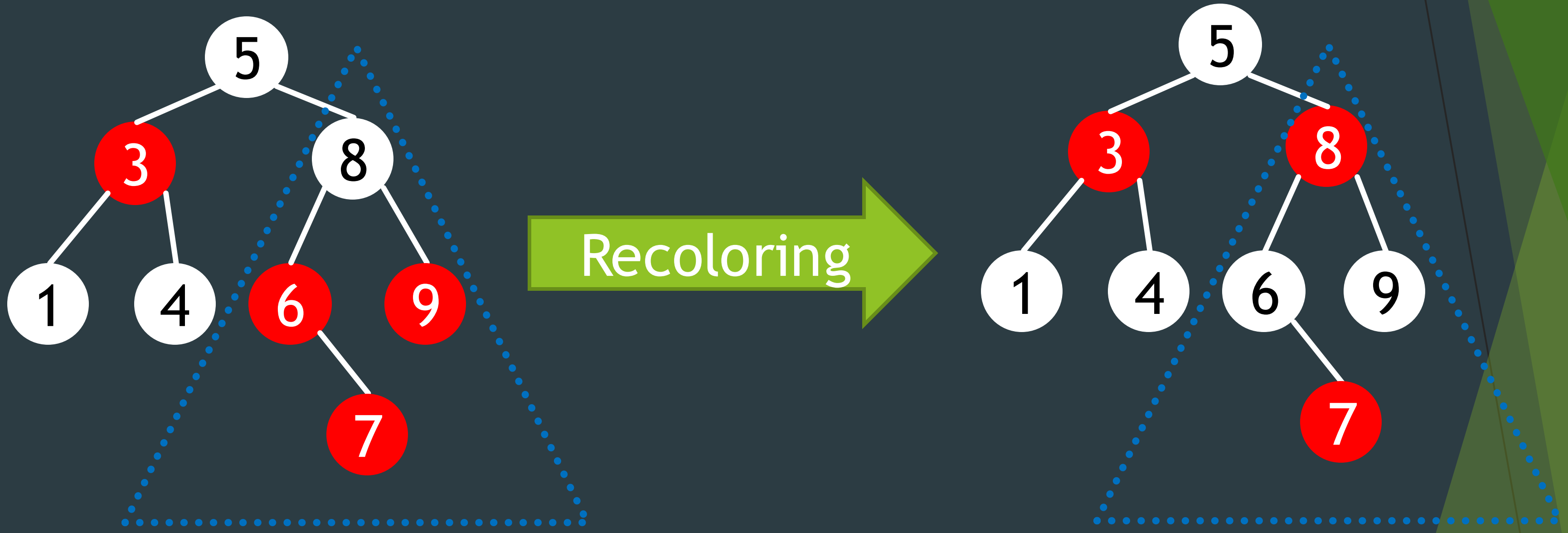
We have to do some work…

Modification: Rotation in AVL Tree

- Maintain the binary search tree property.
- Can be done in $O(1)$ time.



Modification: Recoloring



Invariants

- Red Rule: Red nodes do not have Red children
- **Black Height** Rule (Path Rule): Paths that stem from the same node have the same black heights.

Insertion: Sketch

- Insert x as a leaf.

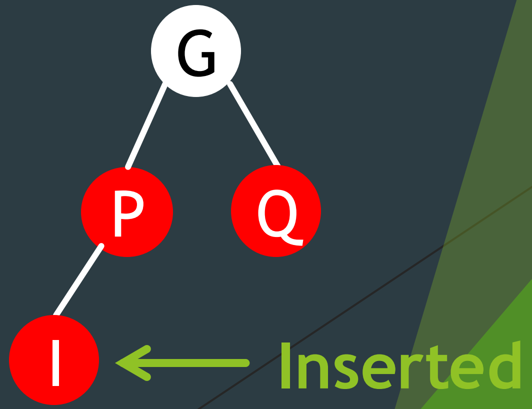
- Color x red.
  - Only red rule may be violated.

- Move the violation up the tree by recoloring/rotation.
  - At some point, the violation will be fixed.

Key idea:
We prioritize the maintenance of the **Black Height Rule** over the **Red Rule**

Violation at Leaf

- **<u>Note</u>**: only red rule may be violated by inserting a (red) node as a leaf.
- When violating, its parent is red and its grandparent is black.
- **<u>Denote</u>**: the inserted node as “I”, its parent as “P”, its grandparent as “G”.



Which Statements Are Correct?

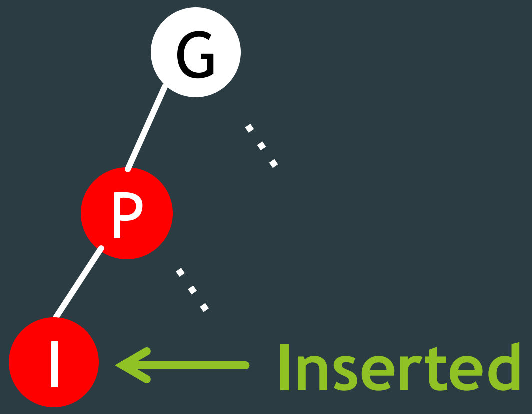
- Suppose there is a violation at the leaf. Suppose the parent of the inserted node is "P". Select all the correct statements.

A. P could be a non-leaf in the original tree.

B. P could have a sibling.

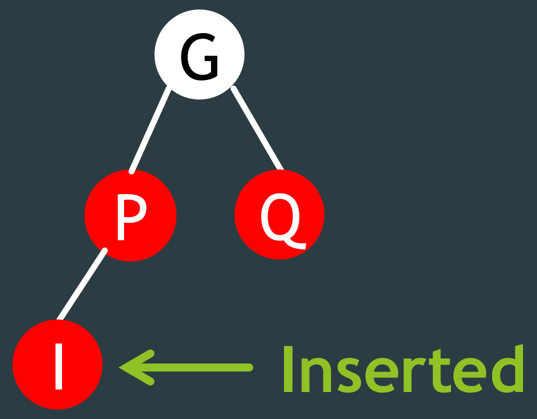
C. P could have no siblings.

D. P could have a sibling and that sibling must be a leaf node.



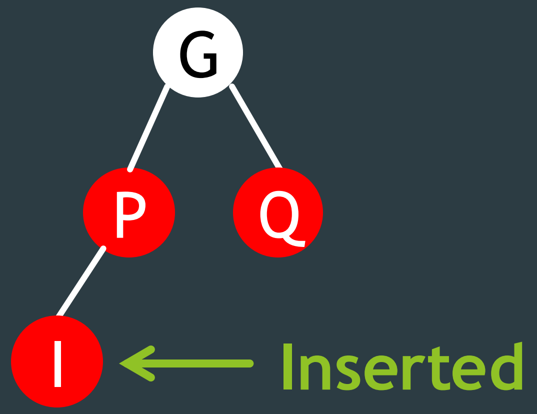
Violation at Leaf

- **Note**: only red rule may be violated by inserting a (red) node as a leaf.
- When violating, its parent is red and its grandparent is black.
- **Denote**: the inserted node as “I”, its parent as “P”, its grandparent as “G”.
- **Claim**: in the old tree, “P” is a leaf, i.e., has no children.



Violation at Leaf

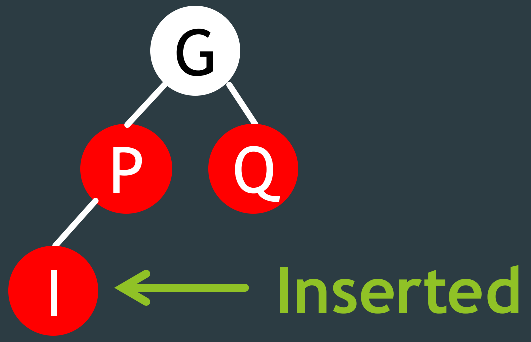
- **Assume**: the parent “P” is the left child of the grandparent “G”.
  - The “right child” case is symmetric.
- **Denote**: the right child of the grandparent to be Q.
- **Claim**: Q is either a red leaf or a NULL.
  - Why?



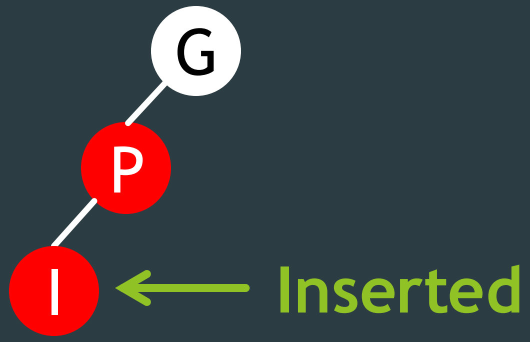
Violation at Leaf

- Three cases:

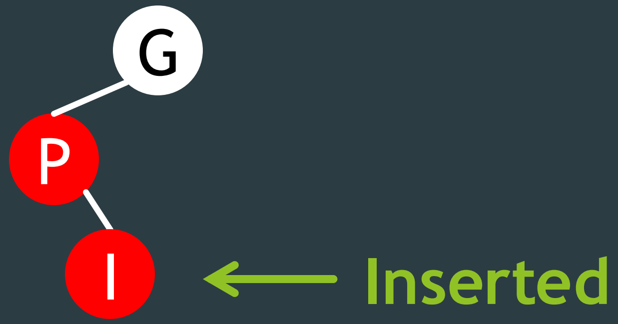
1. Q is a red leaf.



2. Q is empty; I is P’s left child.



3. Q is empty; I is P’s right child.



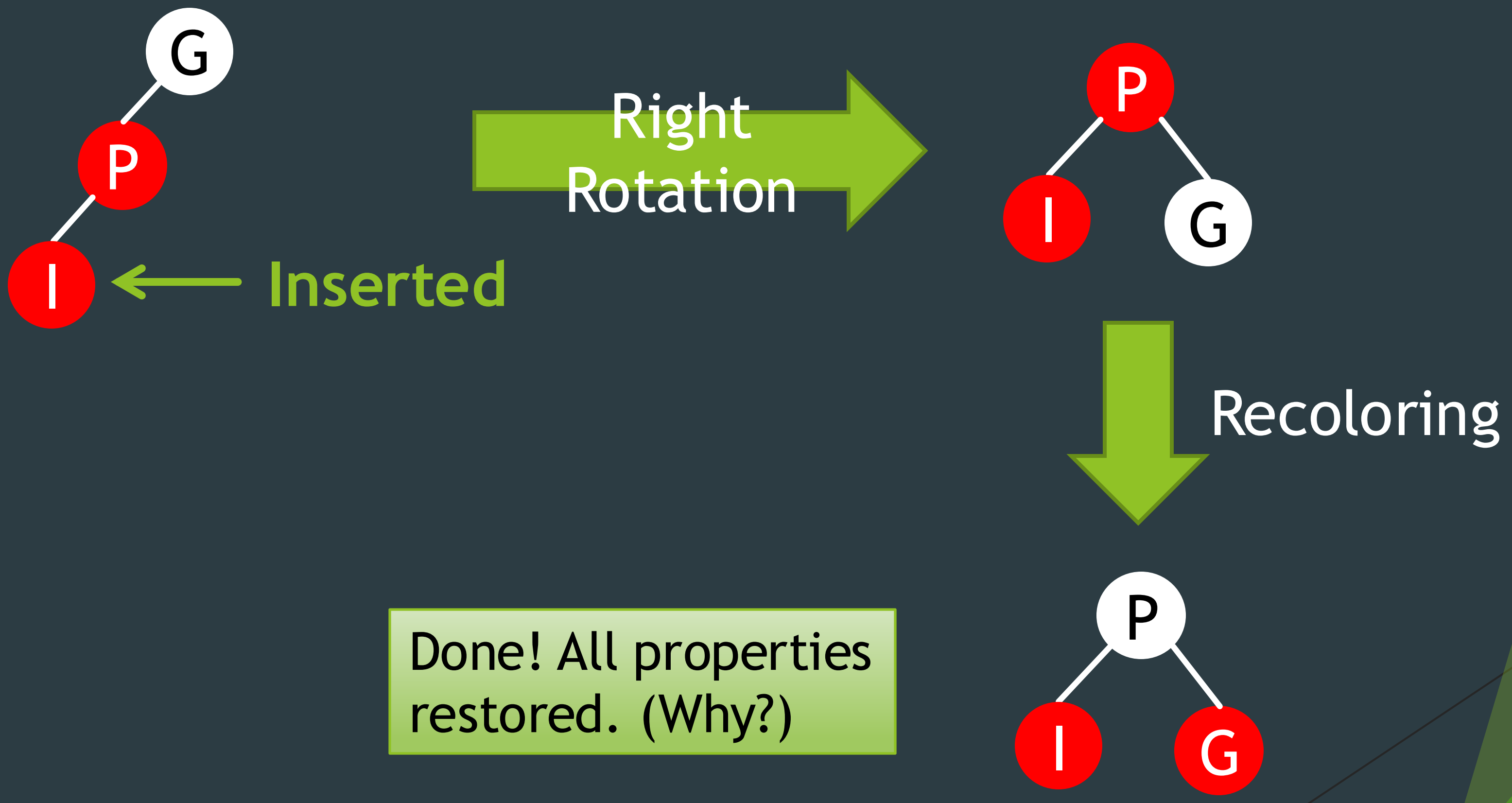
Violation at Leaf

- Case 1: Q is a red leaf.

May **recurse**, since G’s parent may be red.

Violation at Leaf

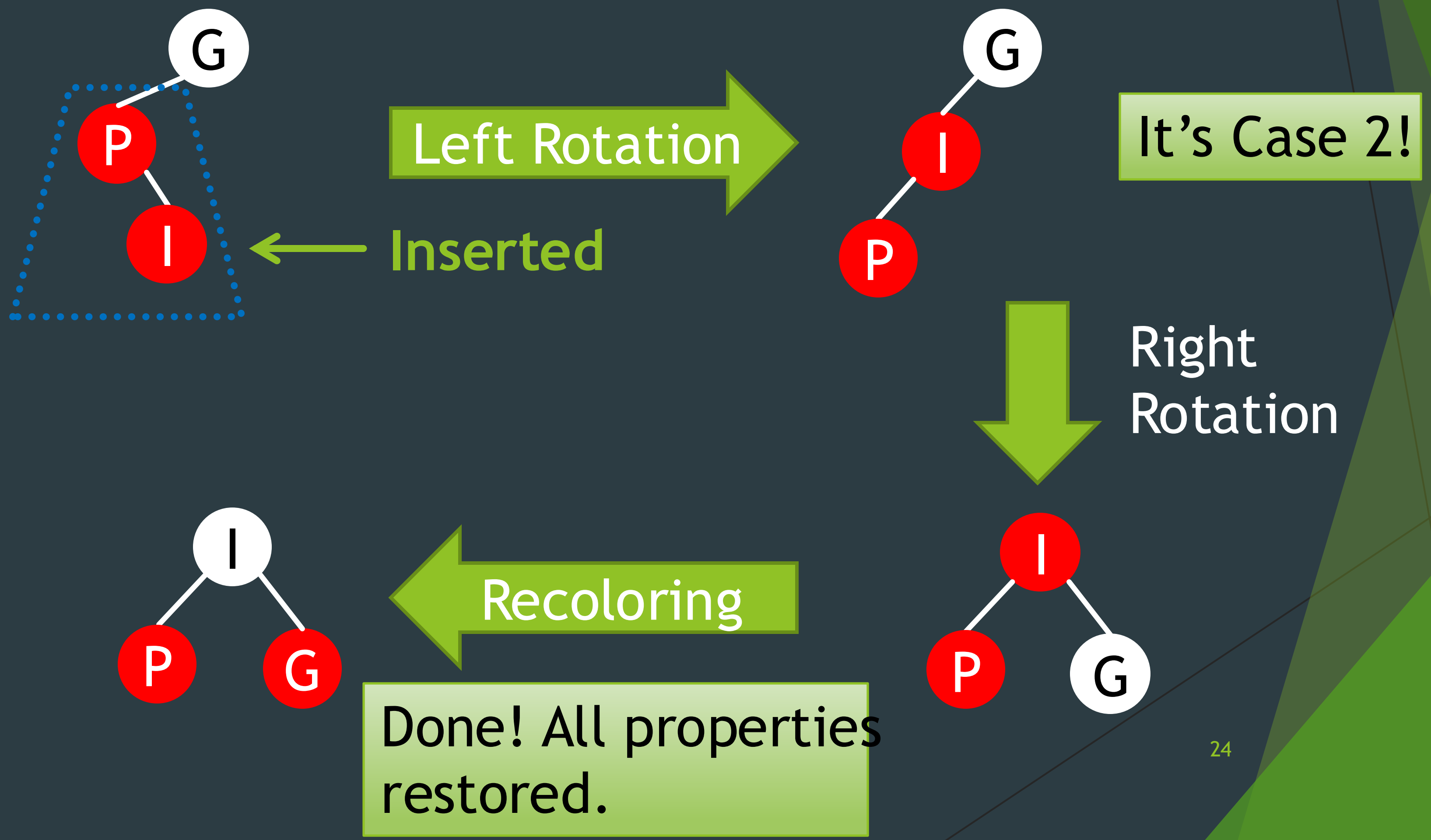
- Case 2: Q is empty; I is P's left child.



Done! All properties restored. (Why?)

Violation at Leaf

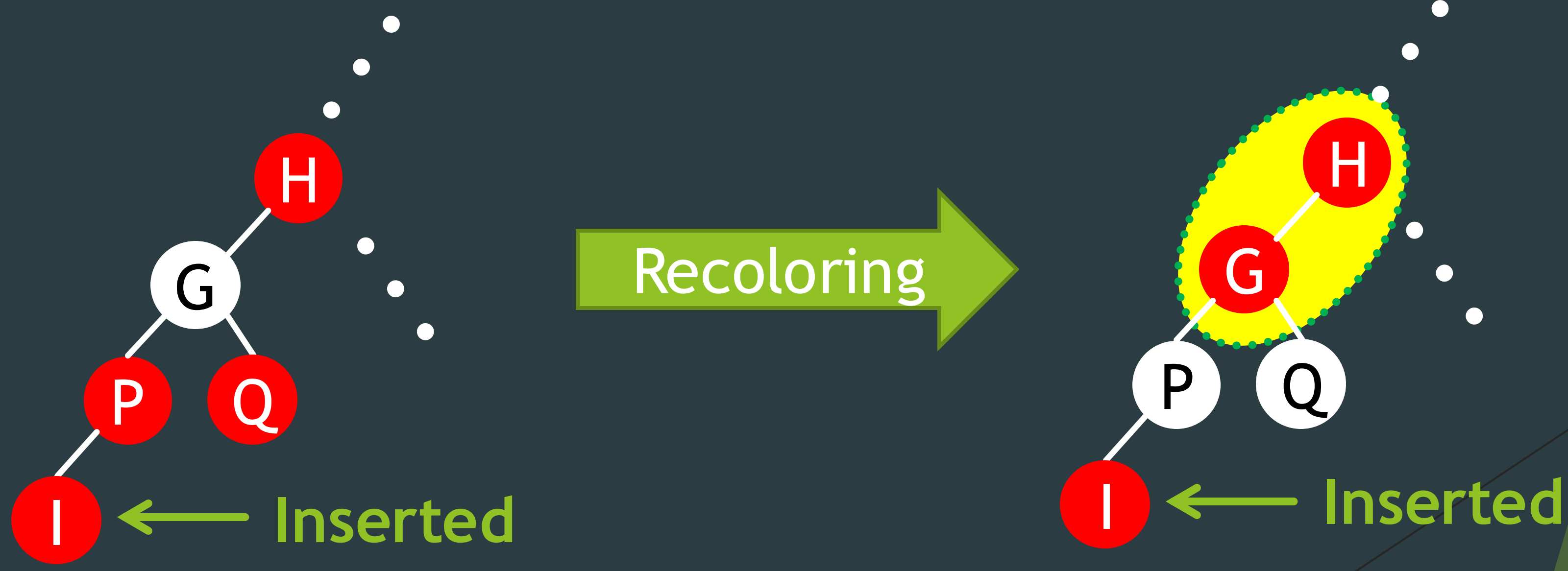
- Case 3: Q is empty; I is P’s right child.



Violation at Leaf: Summary

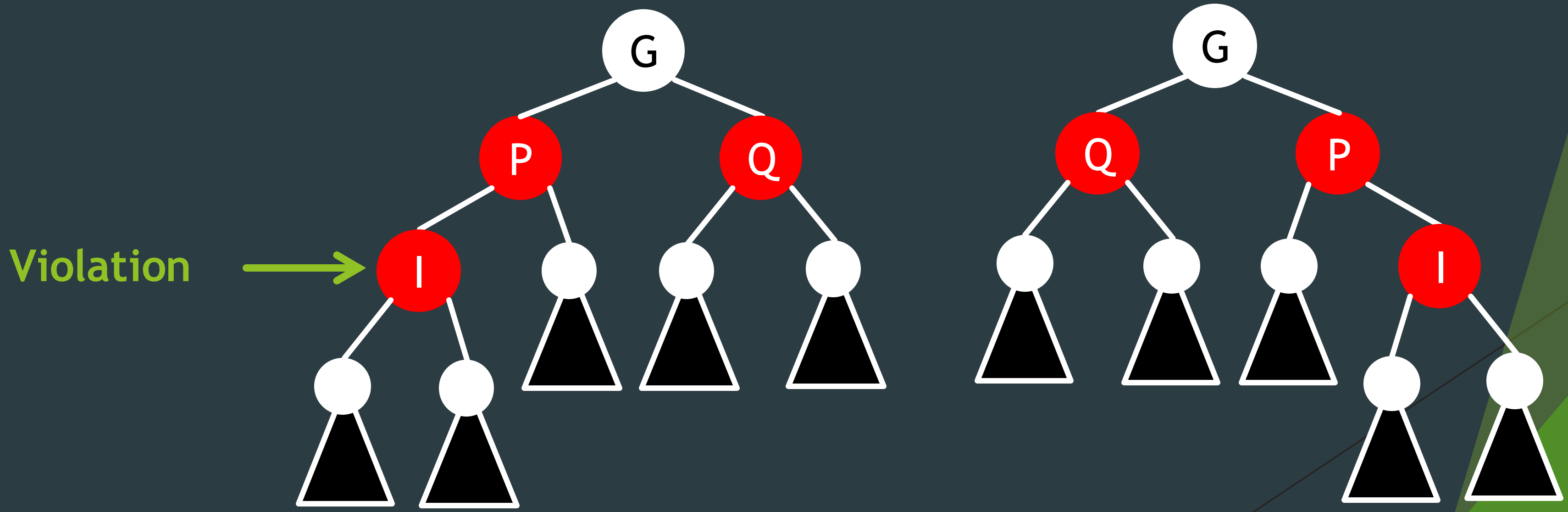
- For Case 2 (Q is empty; I is P's left child) and Case 3 (Q is empty; I is P's right child), we're done.

- For Case 1 (Q is a red leaf), we may recurse.
  - Violation of red rule.



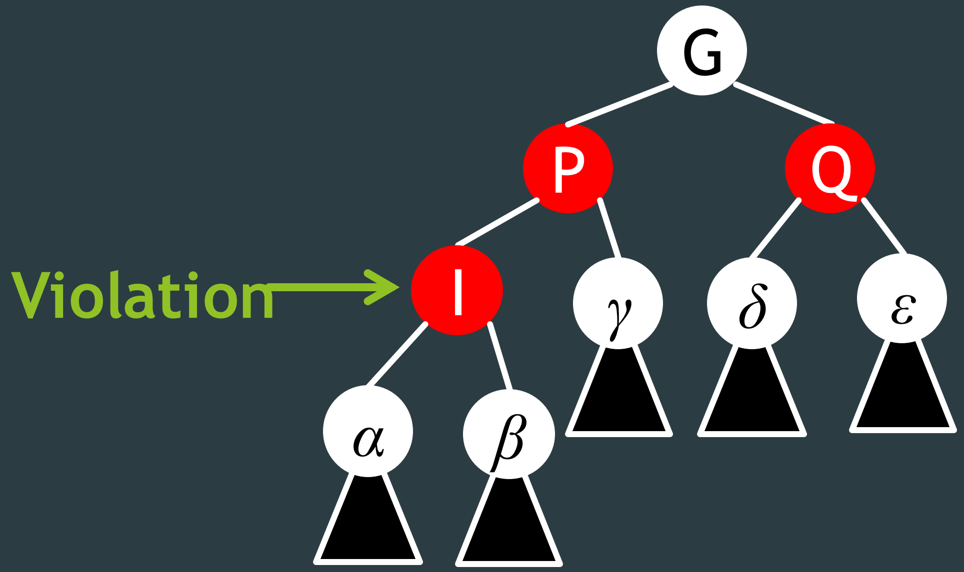
Violation at Internal Nodes

- Caused by moving the violation up the tree.
- When violating, its parent is red and its grandparent is black.
- <u>Assume</u>: the parent “P” is the left child of the grandparent “G”. (The “right child” case is symmetric.)
- <u>Denote</u>: the right child of the grandparent to be Q.



Violation at Internal Nodes

- Three Cases:
  1. Q is a red node.



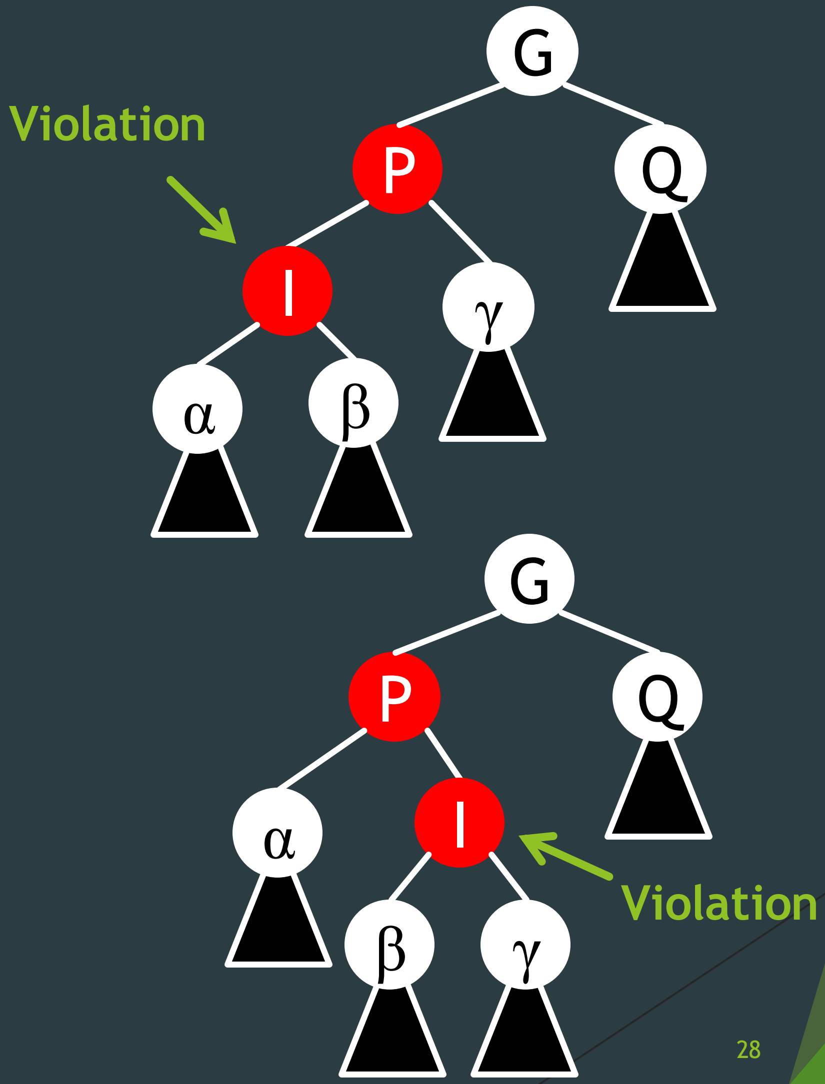
- **Claim**:
  - α, β, γ, δ, ε are trees with **black root**.
  - α, β, γ, δ, ε have the **same** **black height**.

Violation at Internal Nodes

- Three Cases:

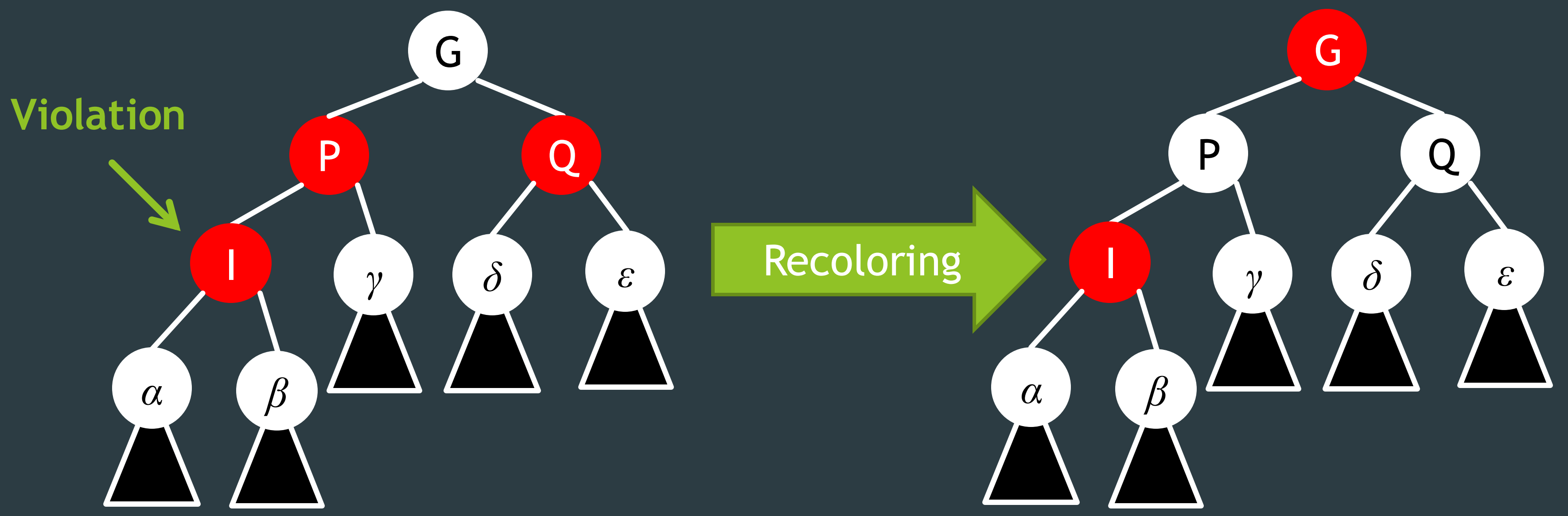
  2. Q is a **black node**; I is P's left child.
  3. Q is a **black node**; I is P's right child.

- <u>Claim</u> for Case 2 and 3:
  - α, β, γ, Q are trees with black root.
  - α, β, γ, Q have the <u>**same**</u> black height.



Violation at Internal Nodes

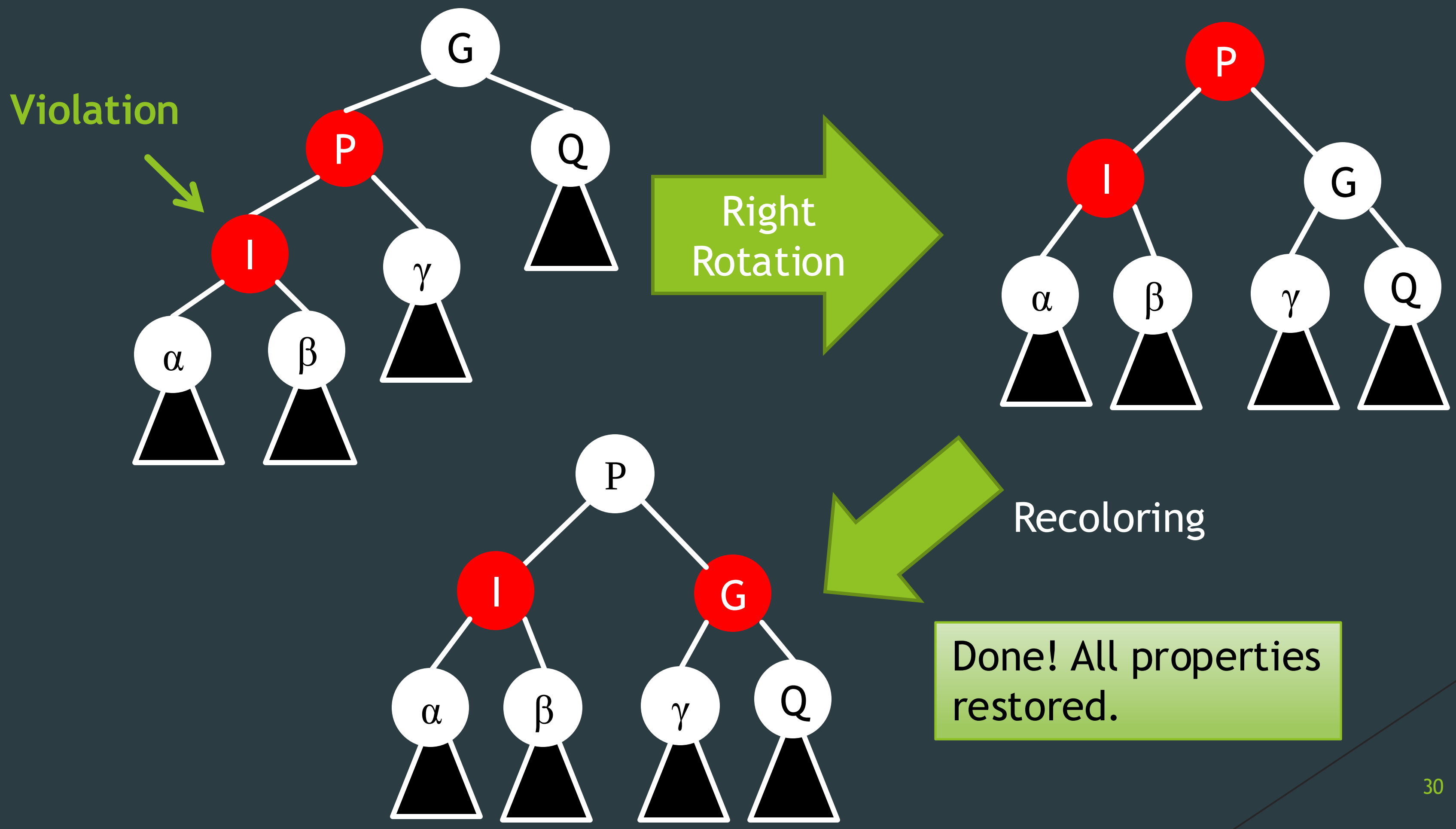
- Case 1: Q is a red node.



May **recurse**, since G's parent may be red.

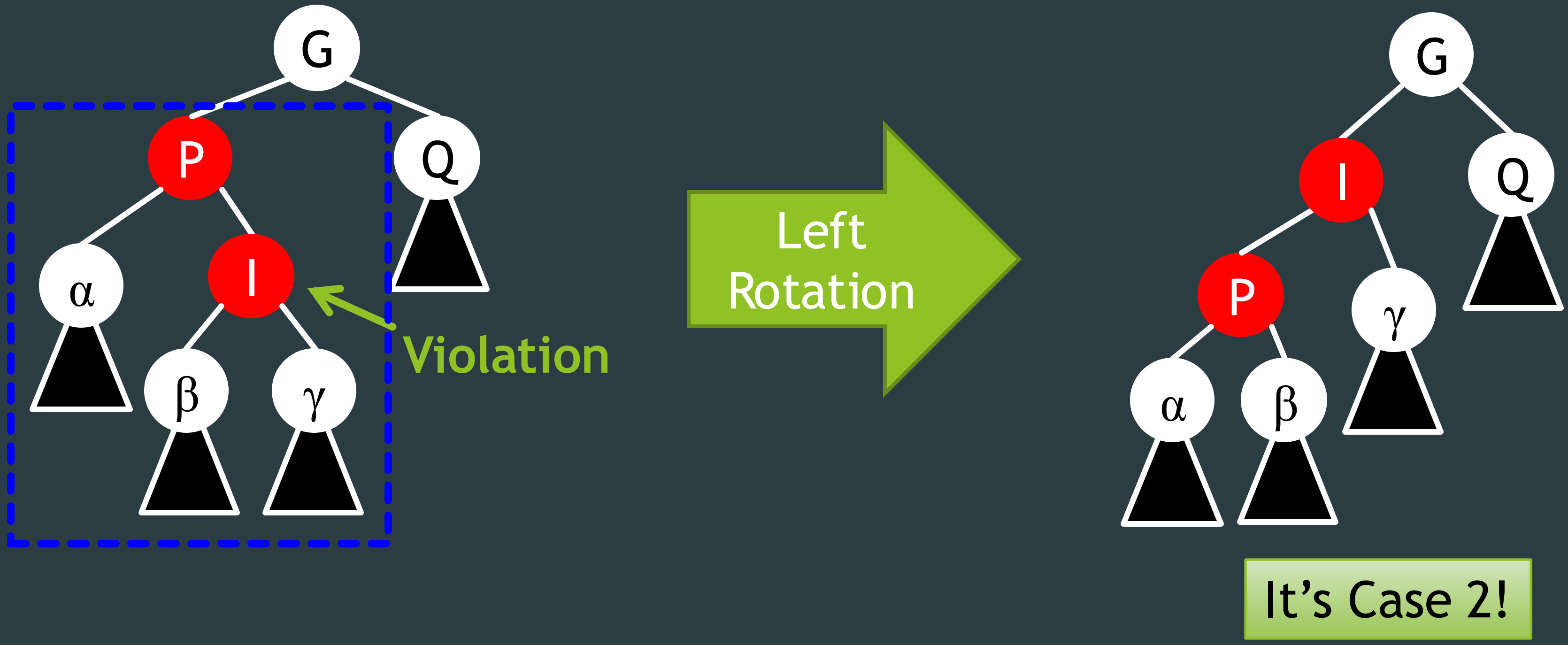
Violation at Internal Nodes

- Case 2: Q is a **black node**; I is P's left child.

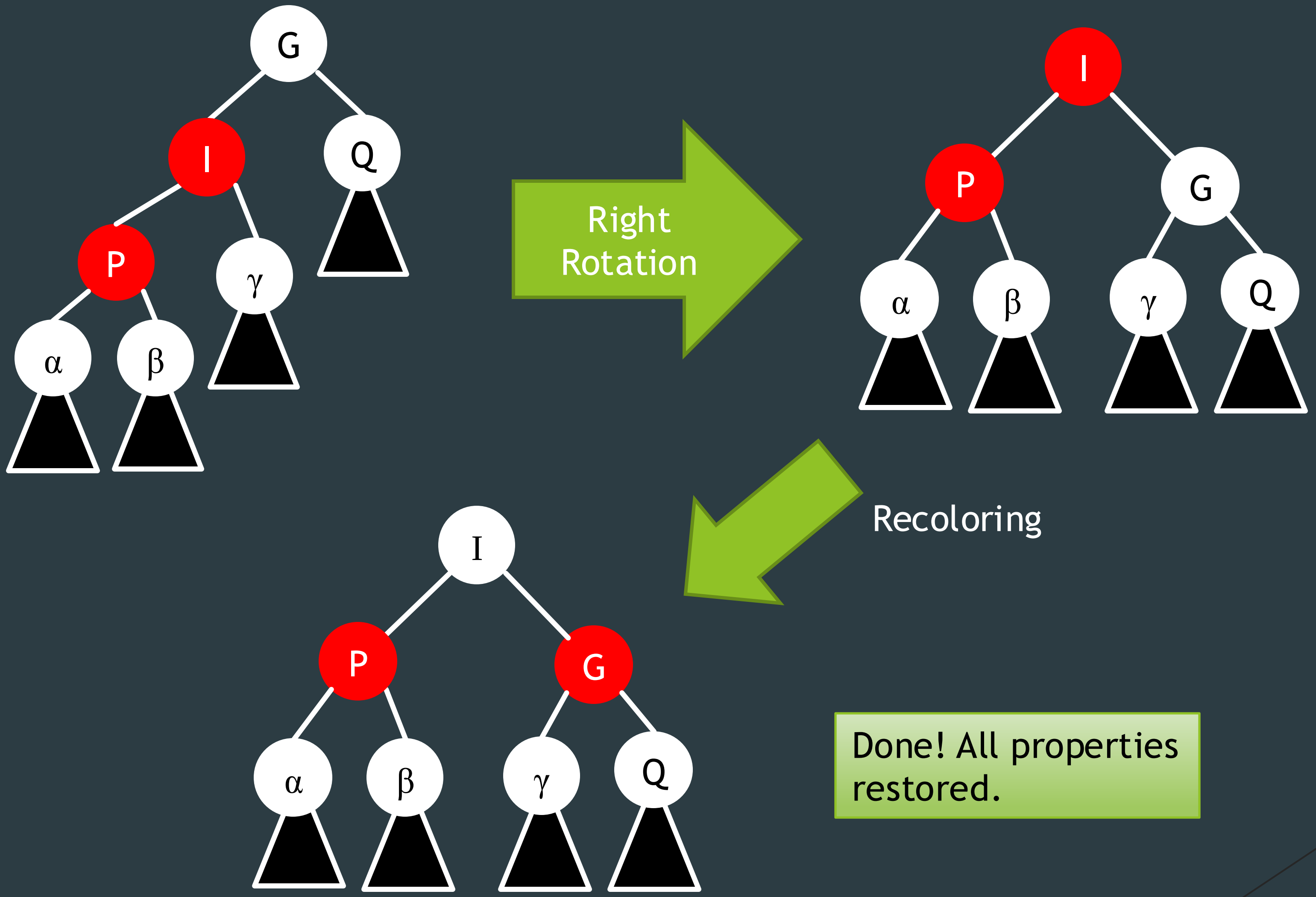


Violation at Internal Nodes

- Case 3: Q is a **black node**; I is P's right child.



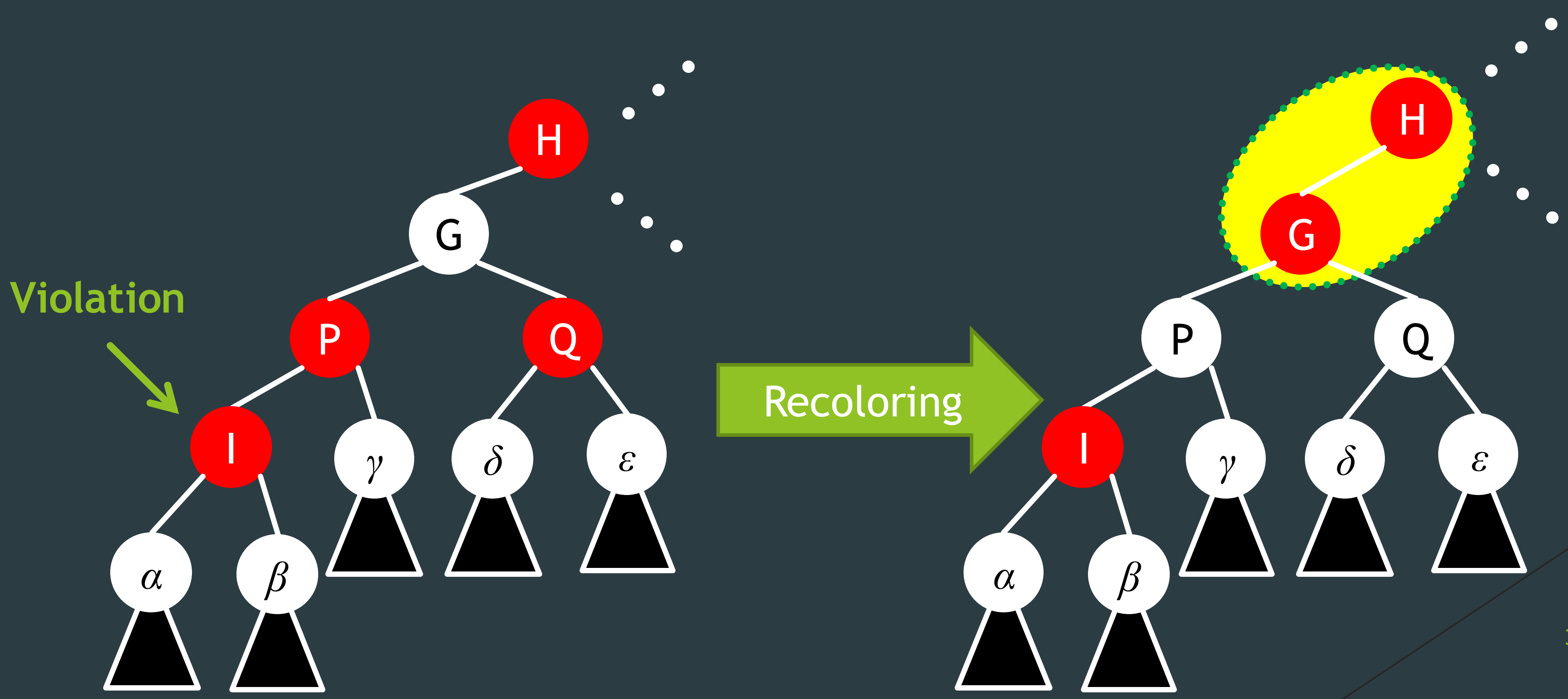
Violation at Internal Nodes: Case 3 (cont.)



Done! All properties restored.

Violation at Internal Nodes: Summary

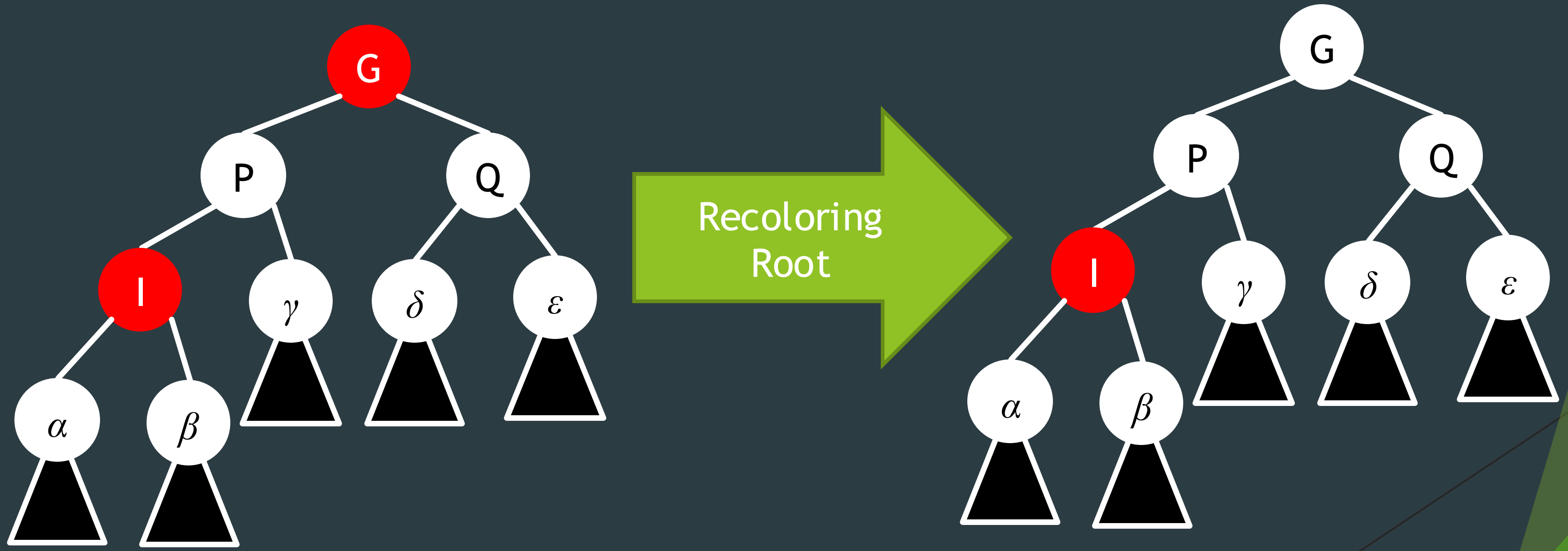
- For Case 2 (Q is a **black node**; I is P's left child) and Case 3 (Q is a **black node**; I is P's right child), we're done.
- For Case 1 (Q is a red node), we may recurse.
  - Violation of red rule.



Final Step: Violation Fix at the Root

- By moving the violation up the tree …
  - … the root may become red.

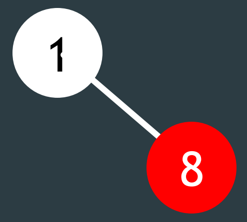
- Final step: set root to be **black**.
  - All red-black tree properties are now **restored**.



Example

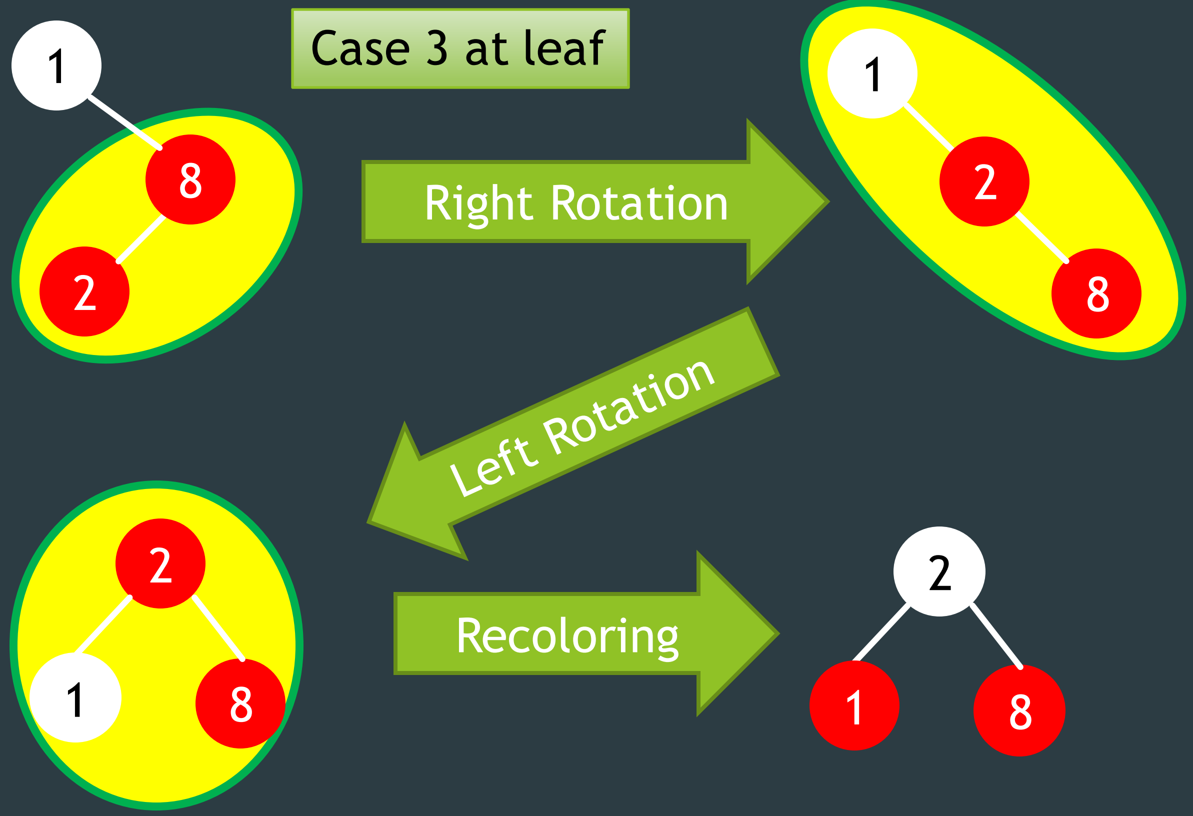
- Insert 1

- Insert 8



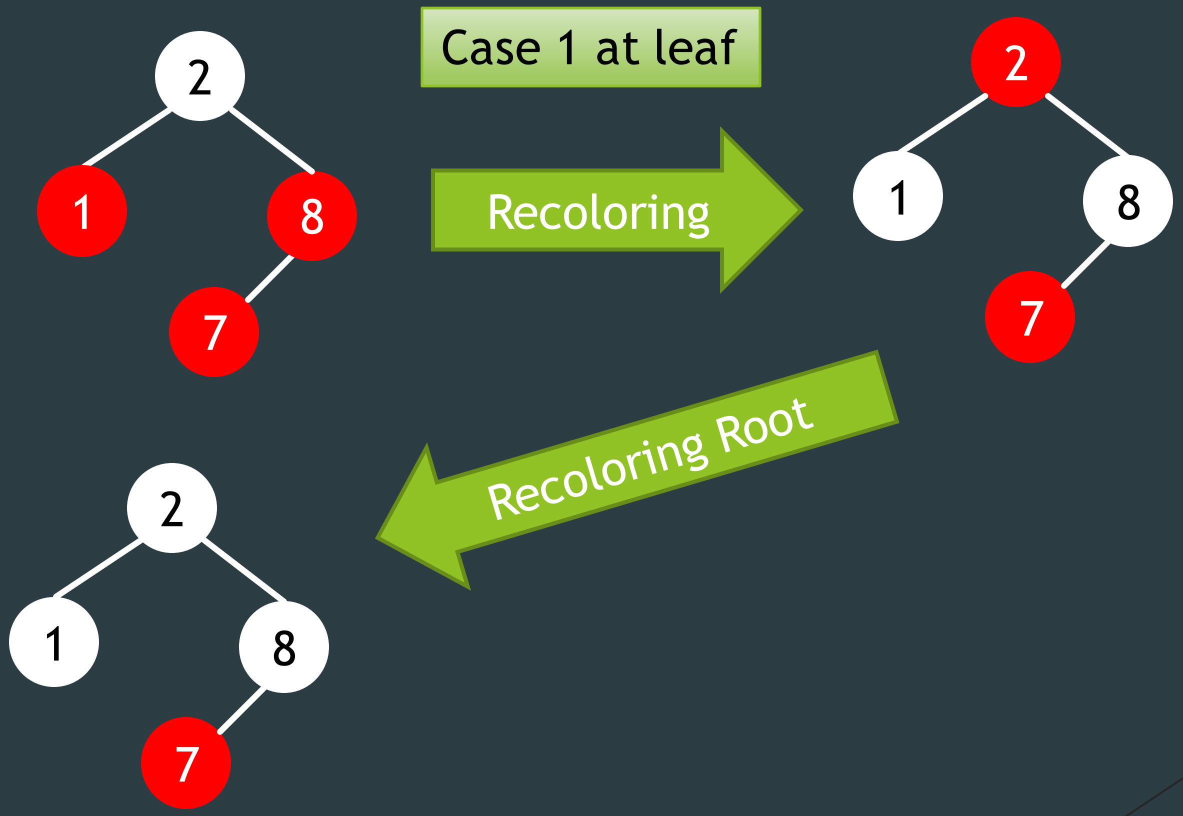
Example (cont.)

- Insert 2



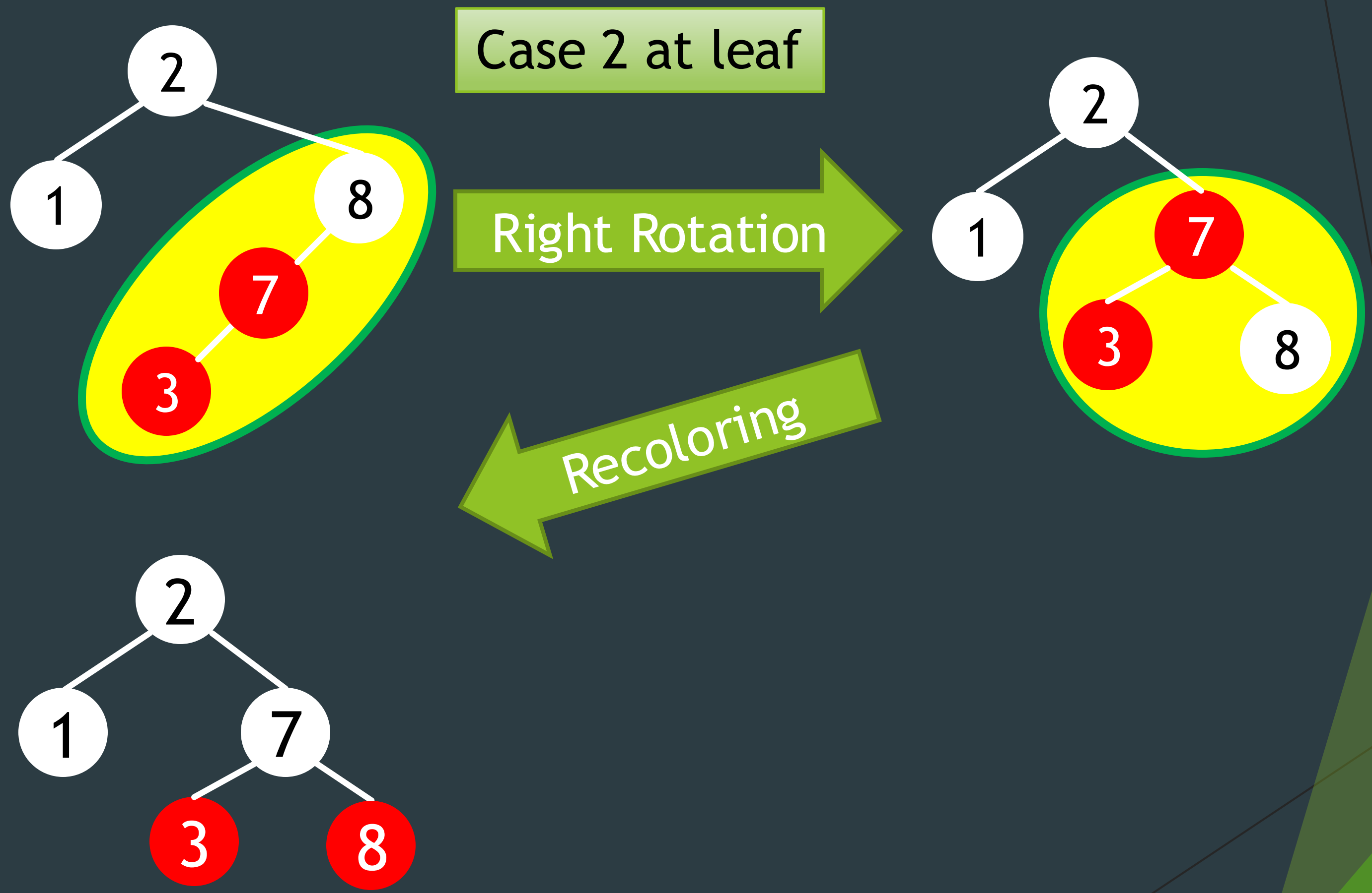
Example (cont.)

- Insert 7



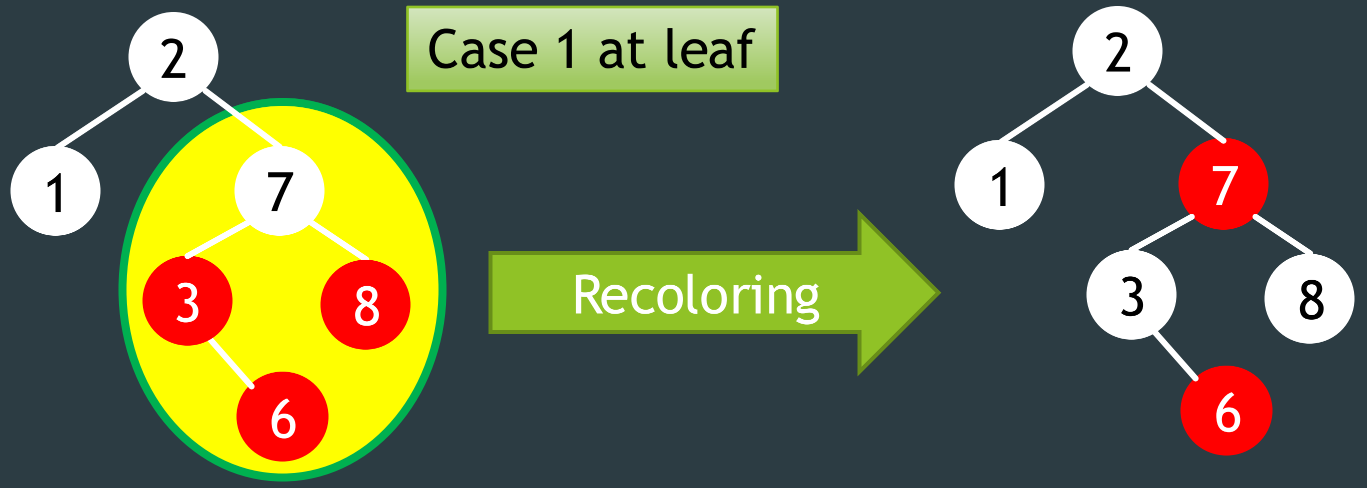
Example (cont.)

- Insert 3



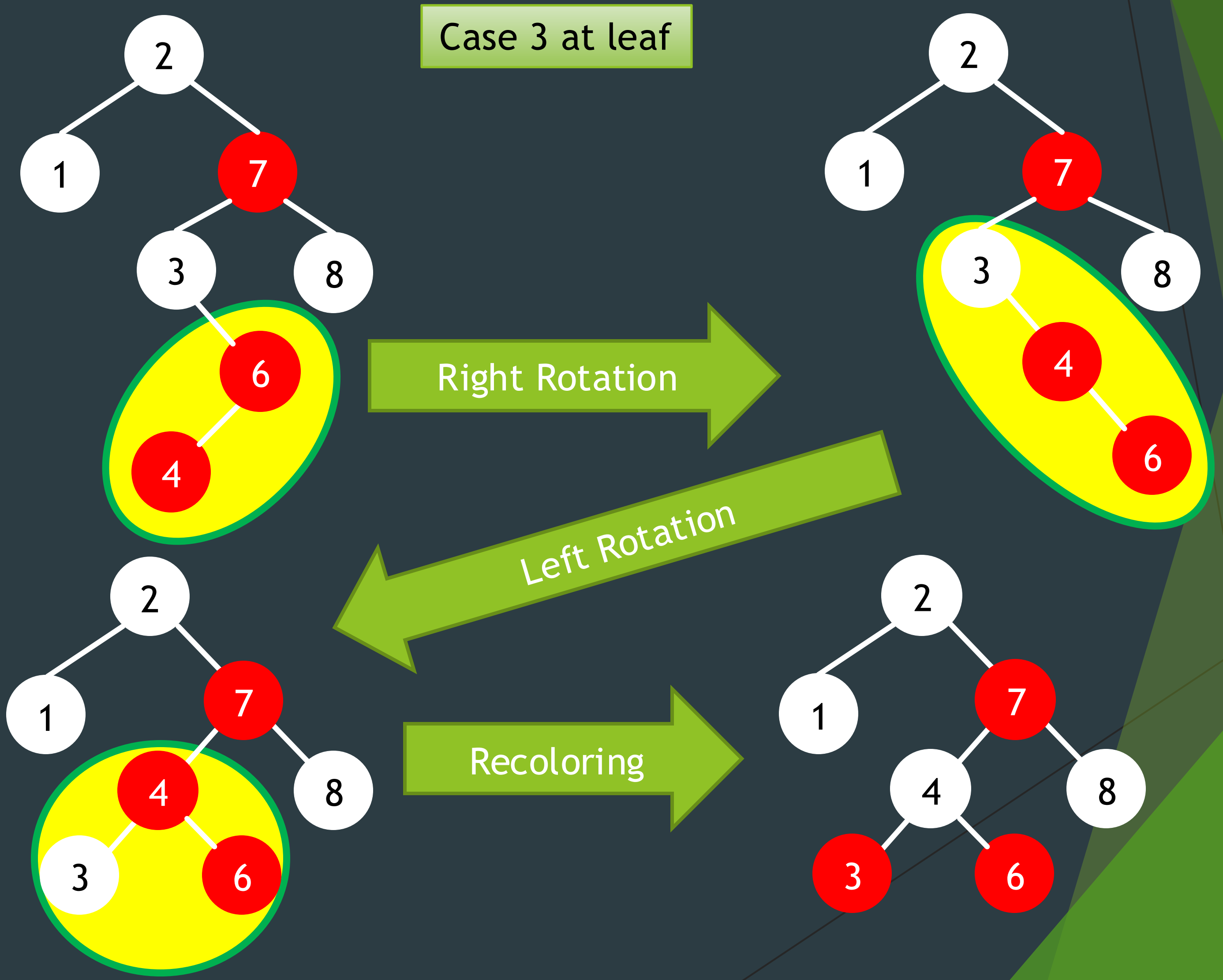
Example (cont.)

- Insert 6



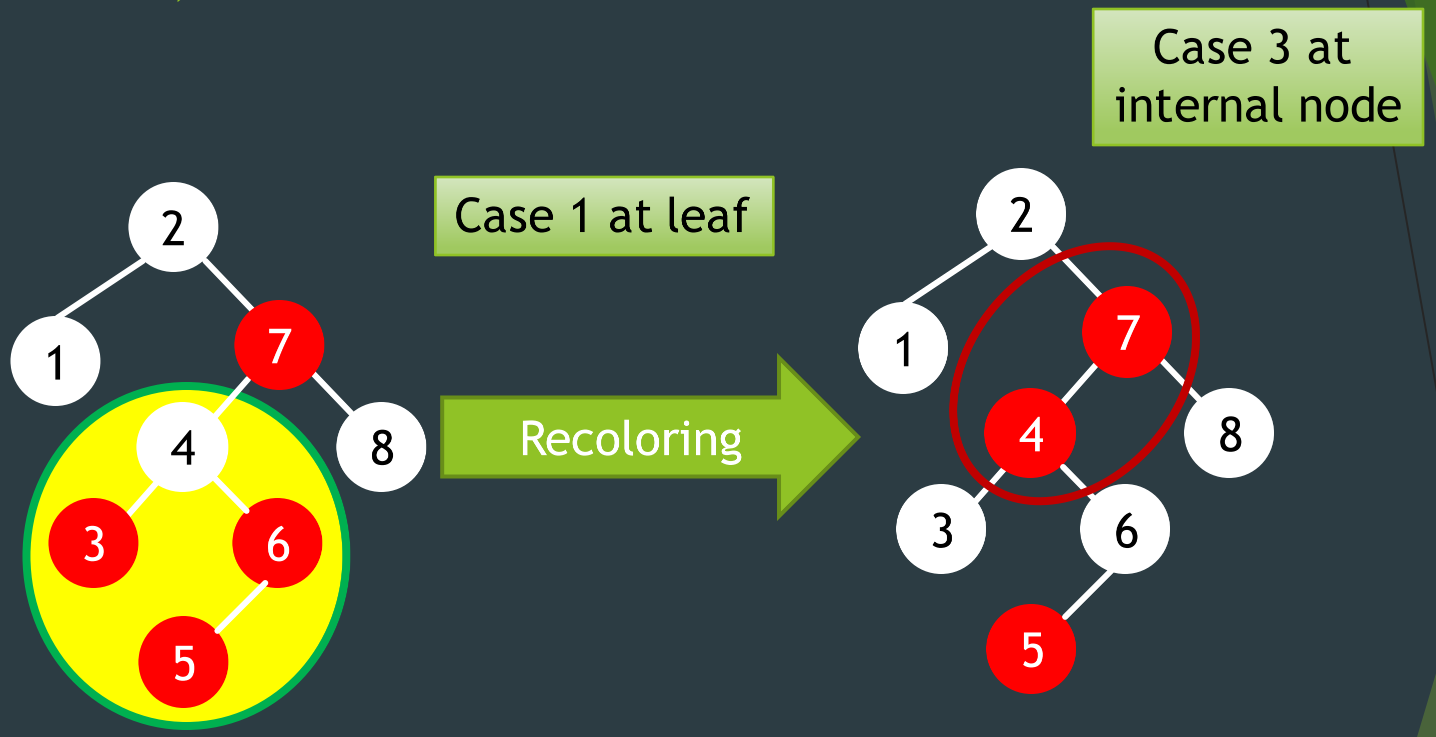
Example (cont.)

- Insert 4

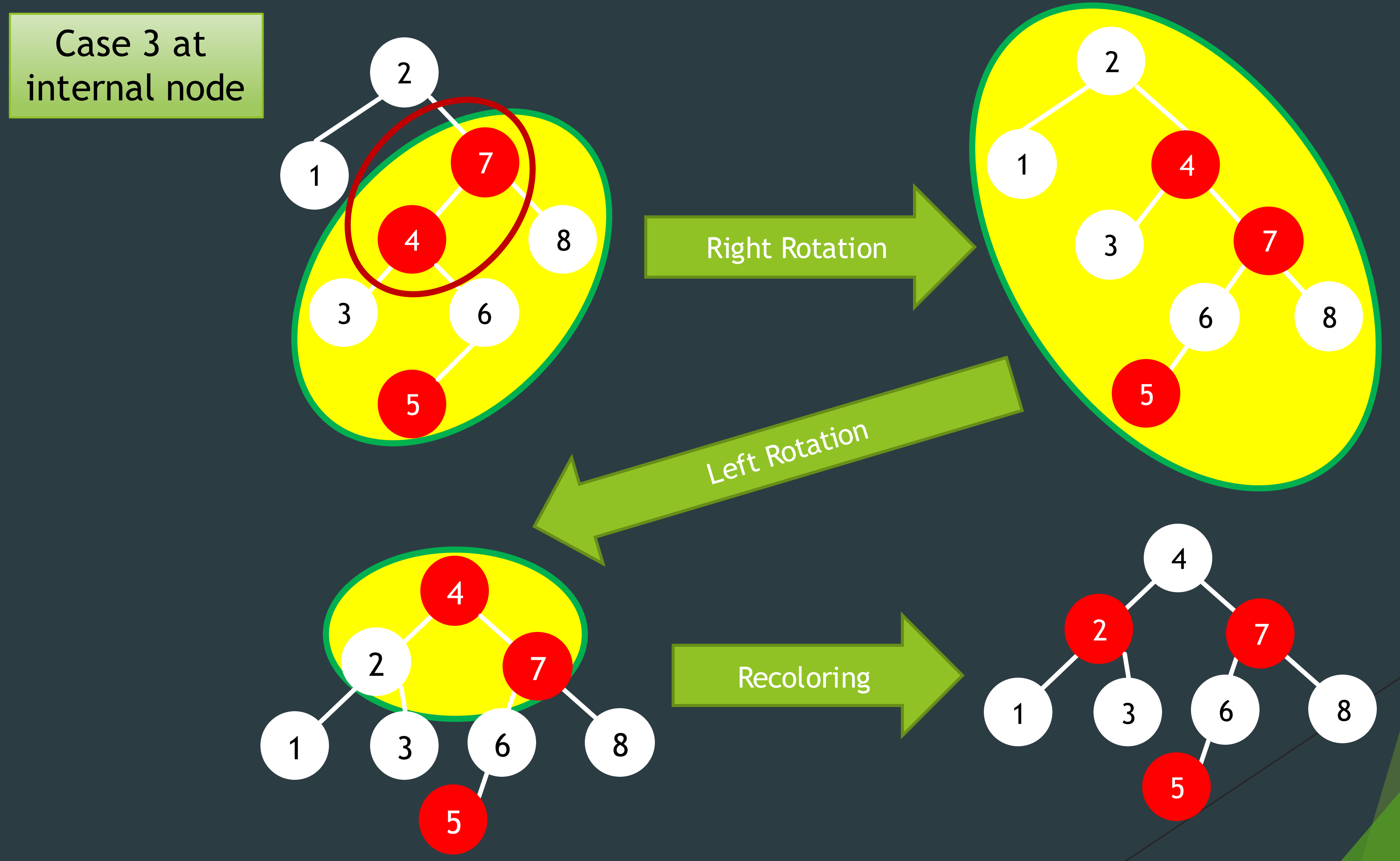


Example (cont.)

- Insert 5



Example (cont.)



Runtime Complexity

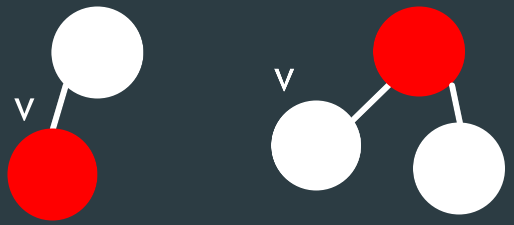
- Number of rotations required
  - For case 1, only need to recolor, no rotation.
  - For case 2 or 3, perform 1 or 2 rotations and terminate.
  - **Thus**: # rotations = $O(1)$.
- Number of recoloring required
  - Worst case: $O(\log n)$
- Runtime complexity is $O(\log n)$.

Compared Against AVL Tree

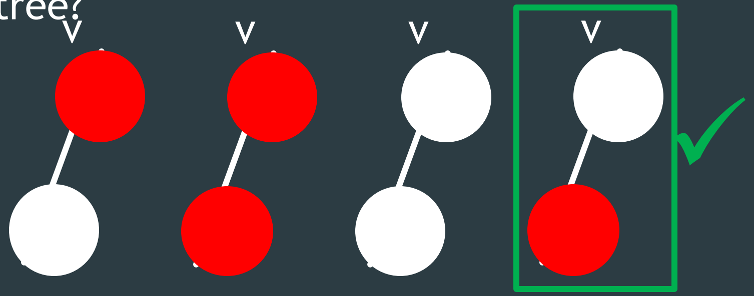
- Tree is less balanced
  - Bad for search
  - Good for insertion/deletion
- What's the best DS for
  - Database (lots of lookups, fewer modifications)?
  - Stock market transactions (lots of modifications)?

Deletion in RB Tree

- What kind of a node is to be removed from RB Tree?
  - Single child or leaf nodes
- What kind of a node could be a leaf node in an RB tree?
  - A red node?
  - A black node?



- What kind of a node would have a single child in an RB tree?
  - A red node?
  - A black node?

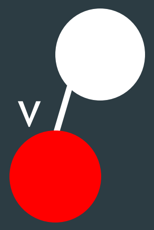


- Any grand children?

Deleting a Red Node

- Simple?

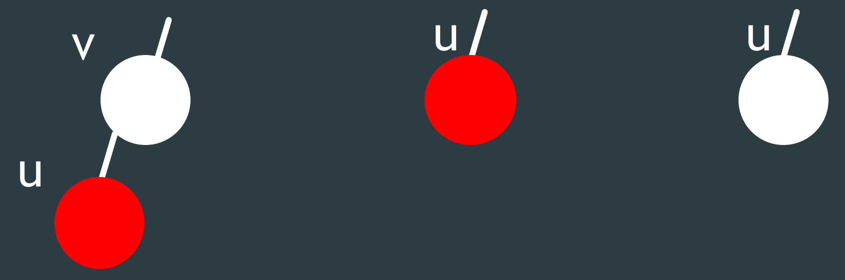
- Simple
  - Just remove it
  - No black height change
  - No red rule violations



Deleting a Black Node

- Simple case:
  - Black node with a red child

- Solution:
  - Delete
  - Recoloring

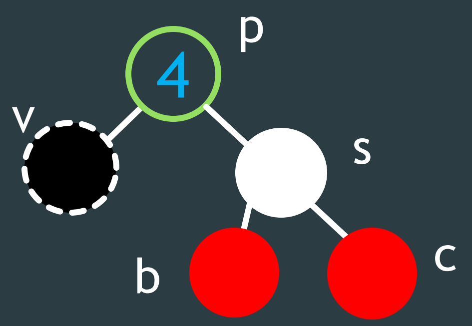


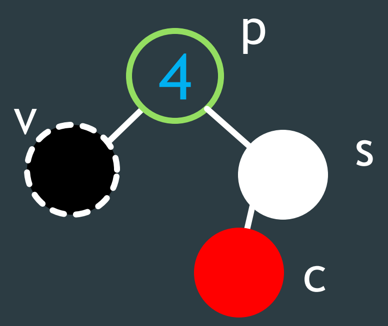
Deleting a Black Leaf

- This is complicated!
  - Black height changes!
    - Reduced by 1
- Fix: somehow retain the black height
  - Fix top: turn a red node to black!
  - Fix bottom: maintain the black path rule downward

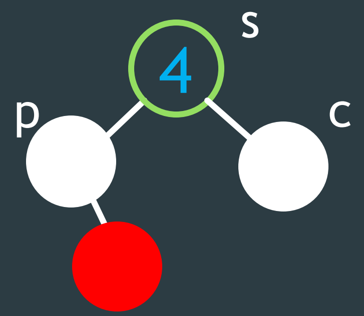
Sibling Has Red Children

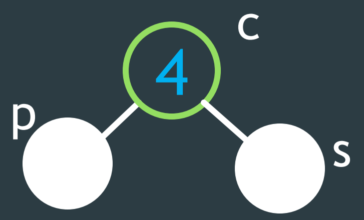
- Sibling has red children:





- We have 3 or 4 nodes left! This means we can rearrange the nodes!





Sibling Has No Red Children

- Case 1: Sibling is black

- Just and recolor the sibling

c

- The end?
  - Nope. The black height of p is reduced! Need recursion!

s

Fix Double-Black

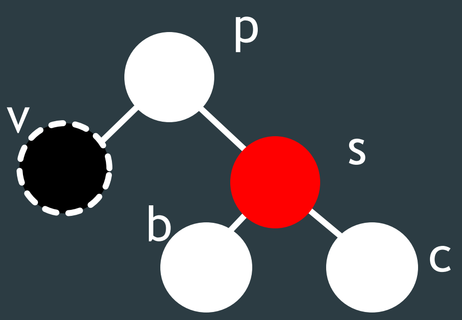
- Consequences with recoloring the sibling:

- Sibling now has the same black height!

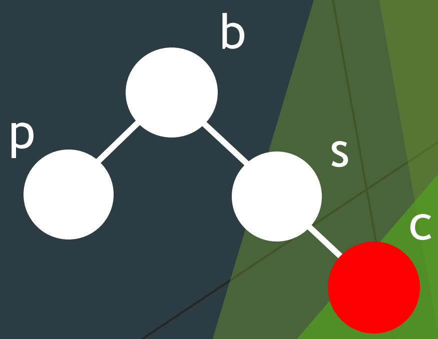
- So… Recurse

Sibling Has No Red Children

- Case 2: Sibling is Red

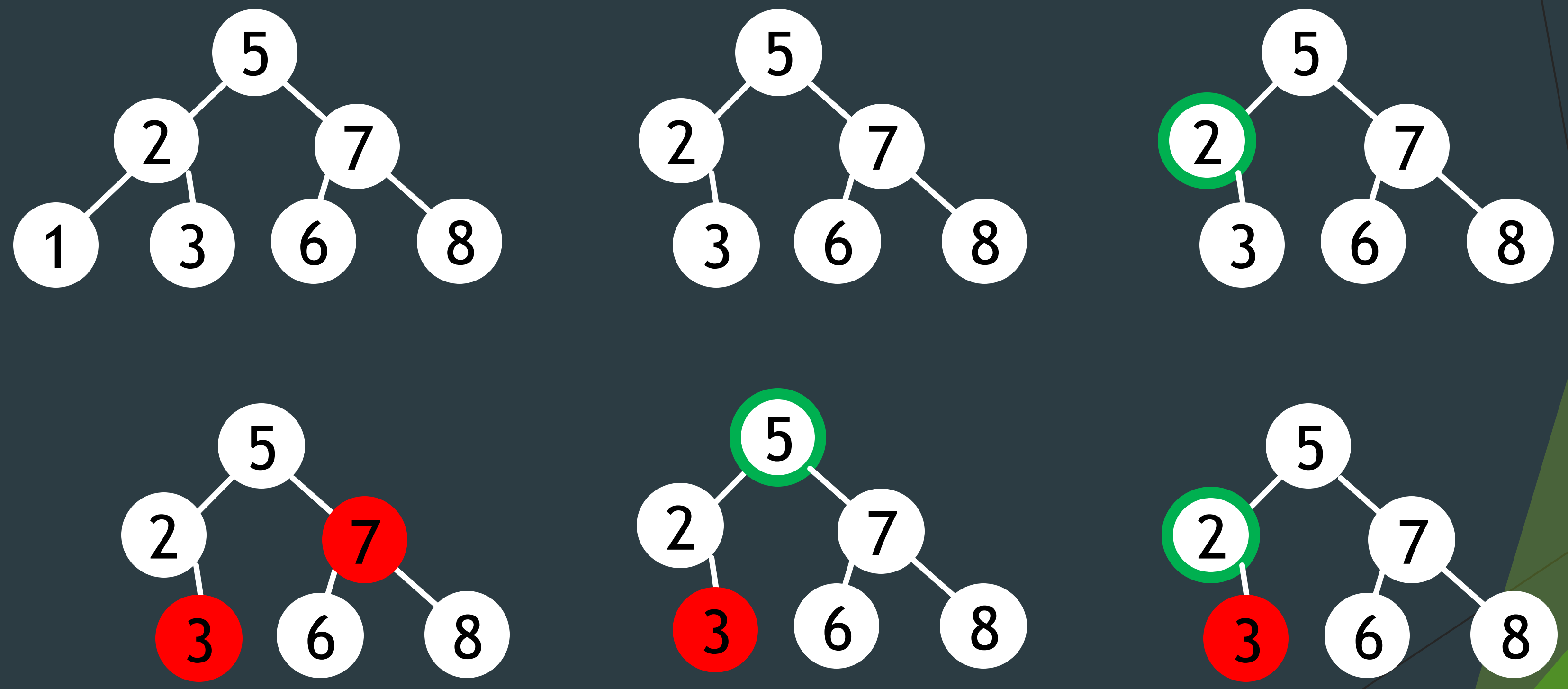


- The sibling must have 2 black children
- After the deletion we will have 4 nodes
- This means we can restructure and recolor once again!

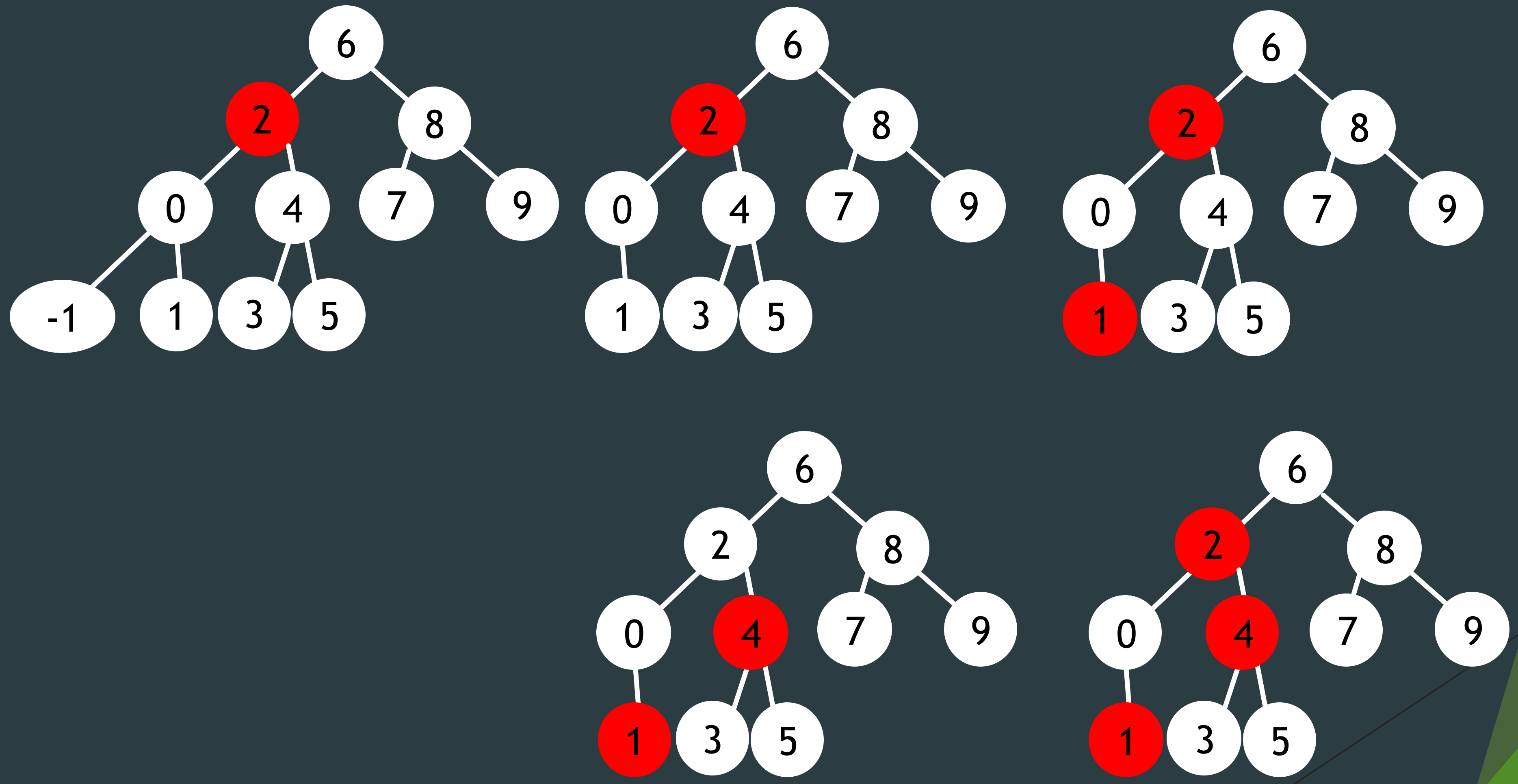


When Does Double-Black Stop

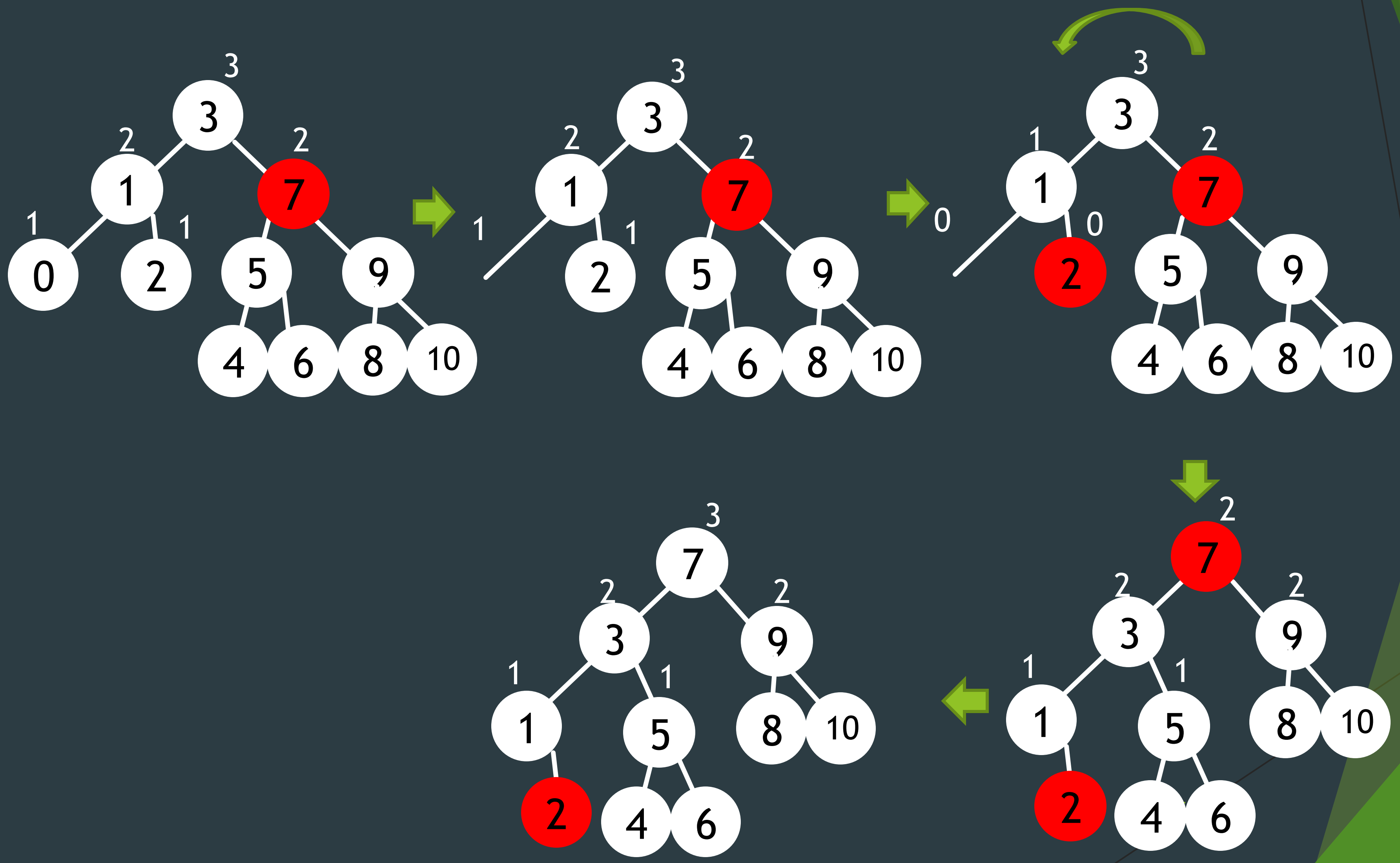
- Until all the way to the root
- Example: delete 1



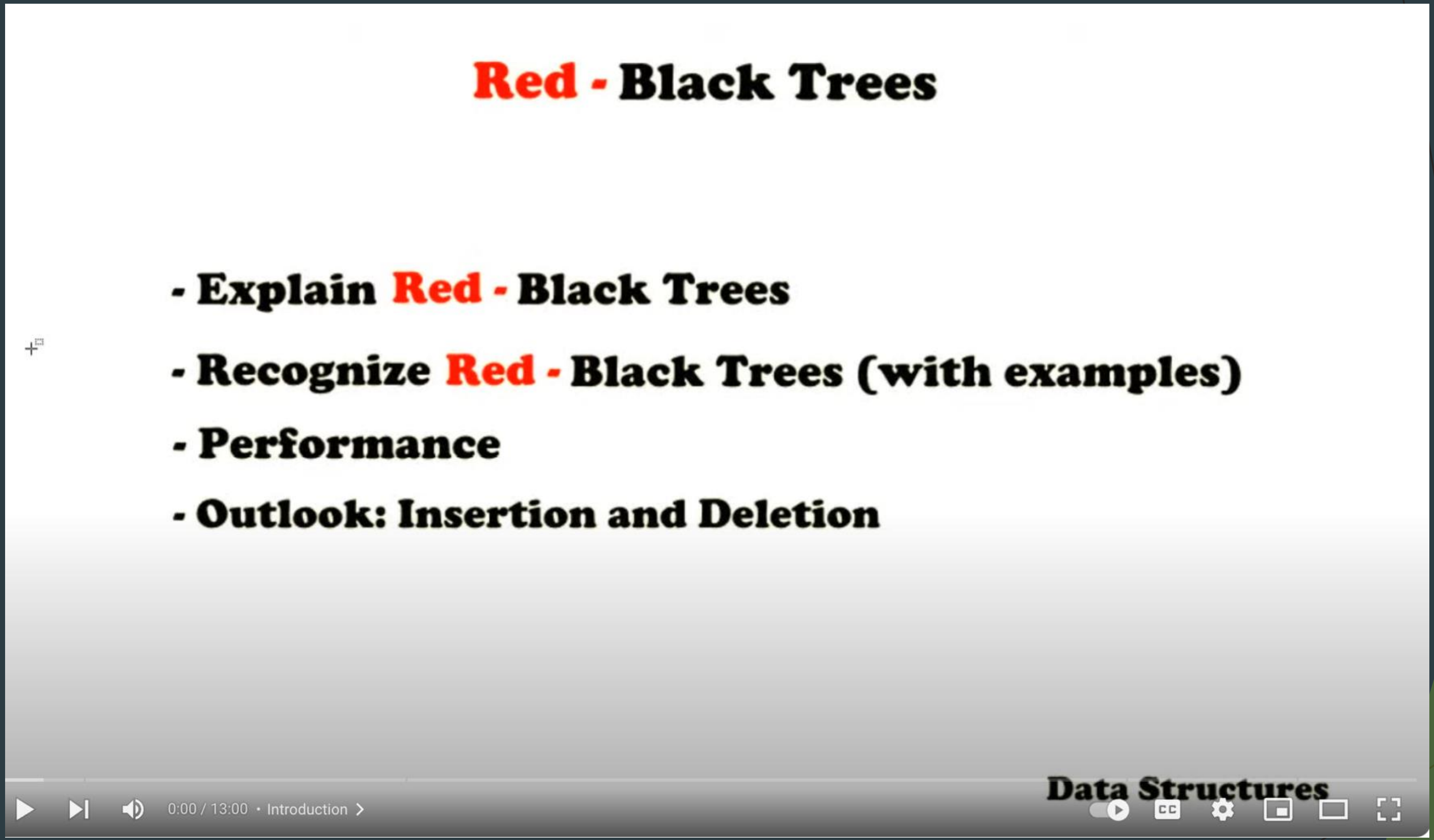
Or No More Double Black + Black Sibling without Red Children



Example #2



Recommended Materials



https://www.youtube.com/watch?v=ZxCvM-9BaXE