Revised: 7/14/2024

Trees II

Michael C. Hackett
Assistant Professor, Computer Science

COMMUNITY COLLEGE OF PHILADELPHIA

Lecture Topics

- AVL Trees
  - Rotation
  - Insertion
  - Removal
- Red-Black Trees
  - Rotation
  - Insertion
  - Removal
- Complexities

AVL Trees

- An **AVL tree** (**Adelson-Velsky and Landis**) is a binary search tree that *self-balances*.
    - If necessary, the tree reorganizes its nodes upon insertion and removal.

- Recall that a tree is balanced if, for any node in the tree, the height of its left subtree and the height of its right subtree differs by 0 or 1.

AVL Trees

- AVL trees do not guarantee to use the minimum height possible.
- It only ensures that, when inserting and removing nodes, the tree remains balanced.

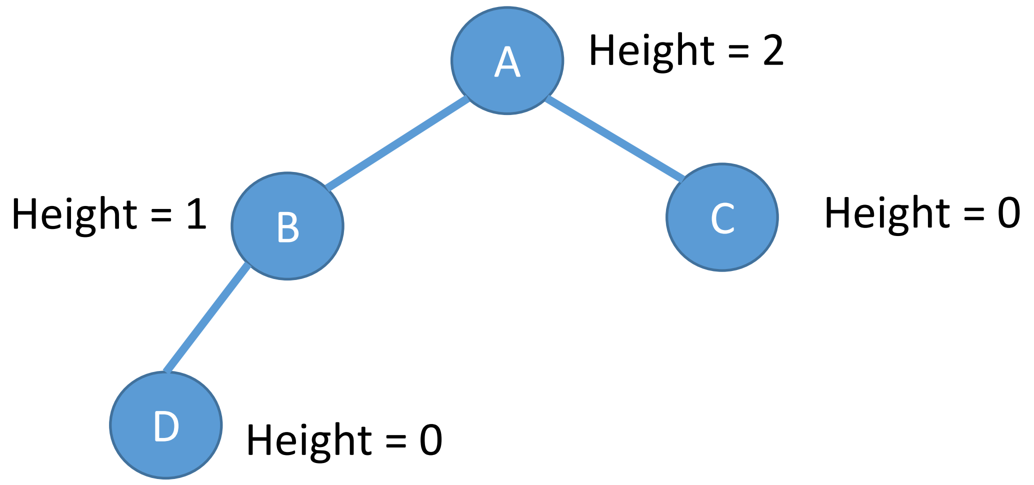
AVL Trees

- A node's **balance factor** is calculated by subtracting the right subtree's height from the left subtree's height.
  - **Balance factor = H_L – H_R**
  - For a balanced tree, the balance factor for each node must be 1, 0, or -1

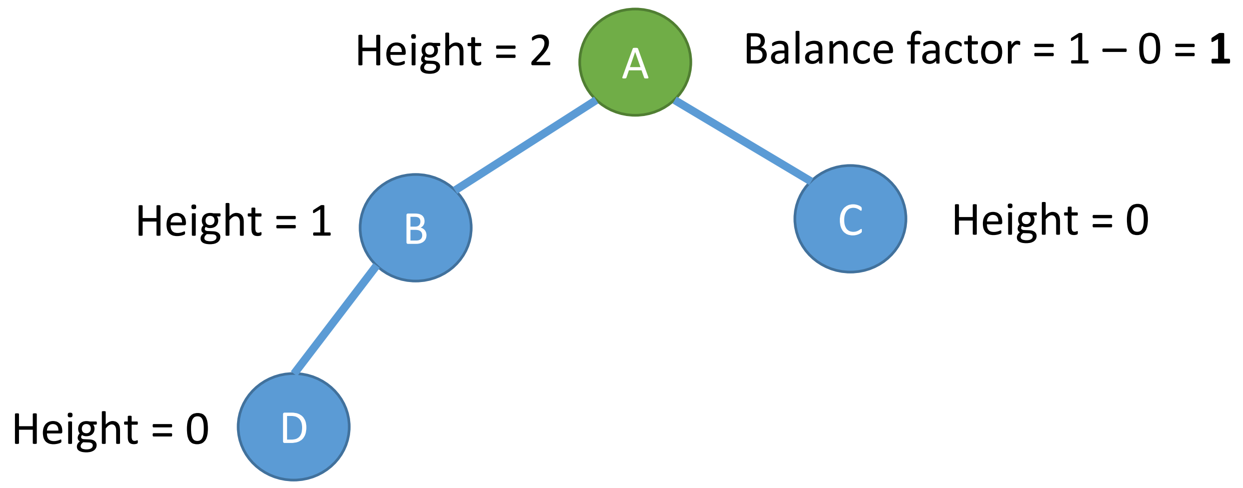
- A tree or subtree that is only one node has a height of 0.
  - In other words, **a leaf has a height of 0**
- **The height of an absent child is -1**

AVL Trees

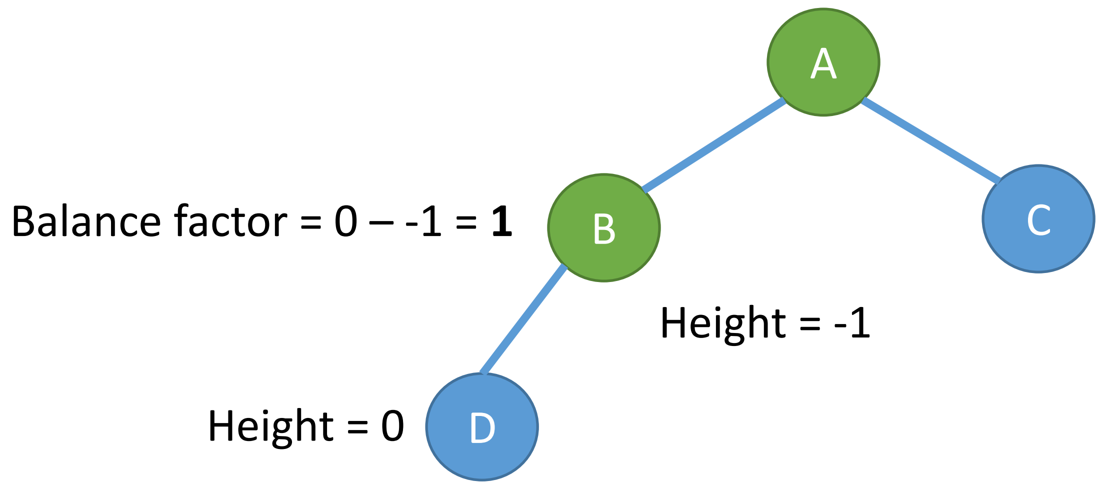
- Nodes in an AVL tree remember their height.
  - Stored in an int field
- This makes it easy (O(1)) to calculate the balance factor of any node.
- When we insert a new node, we add one (if necessary) to each node's height as we traverse back up to the root.



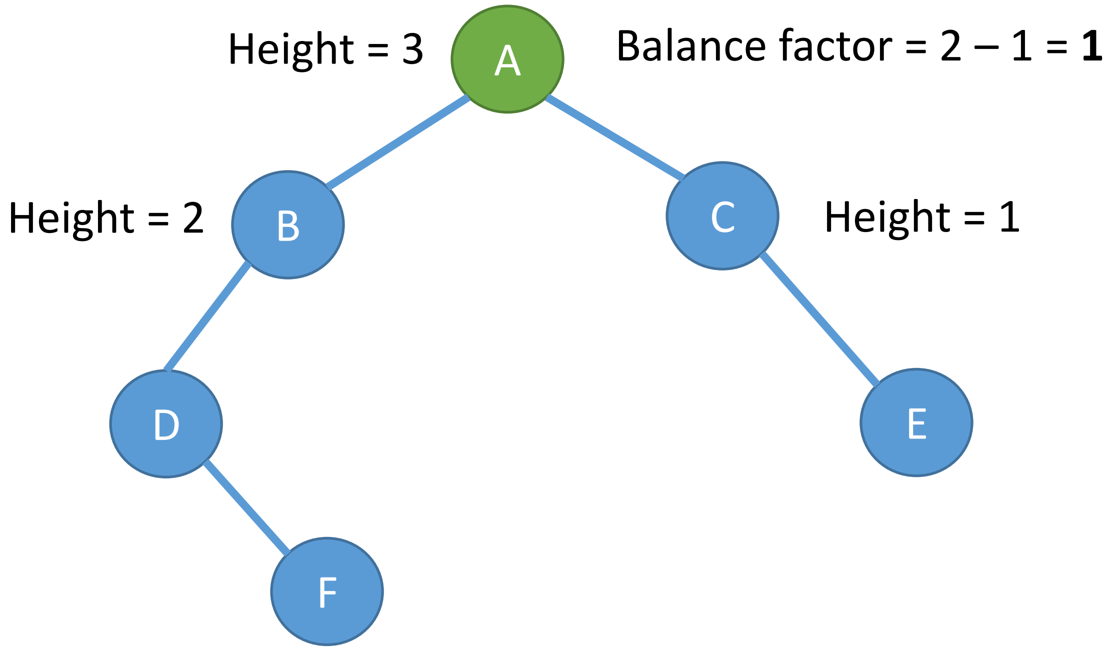
AVL Trees



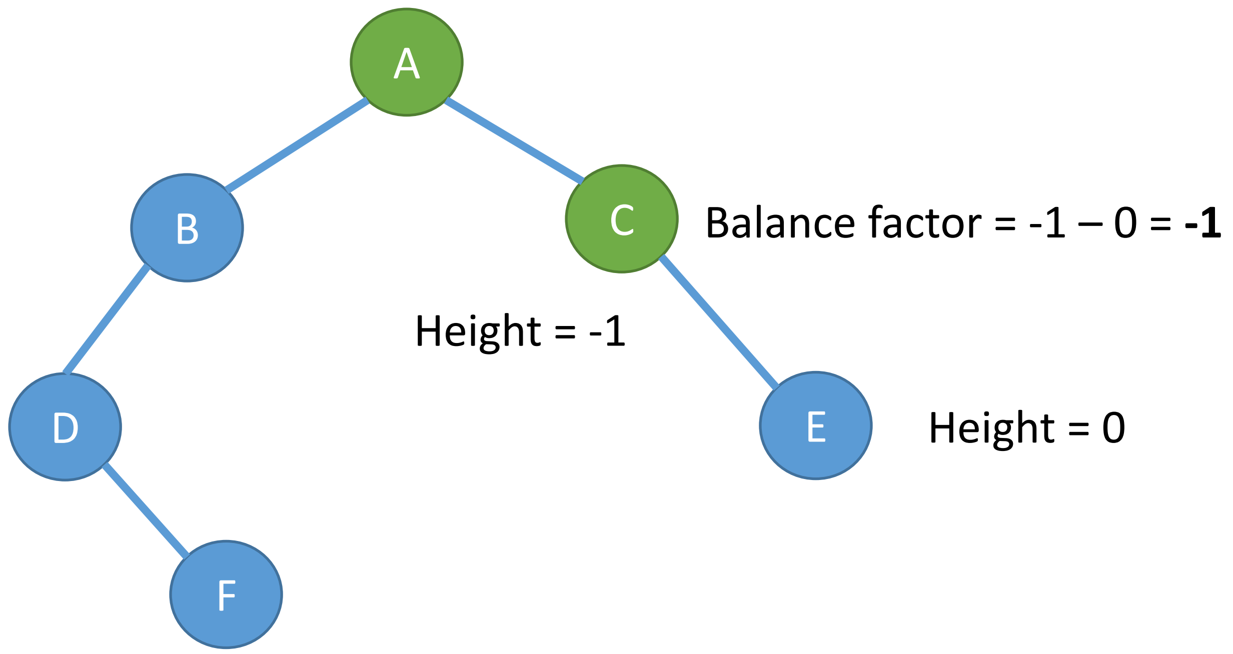
AVL Trees



AVL Trees



AVL Trees



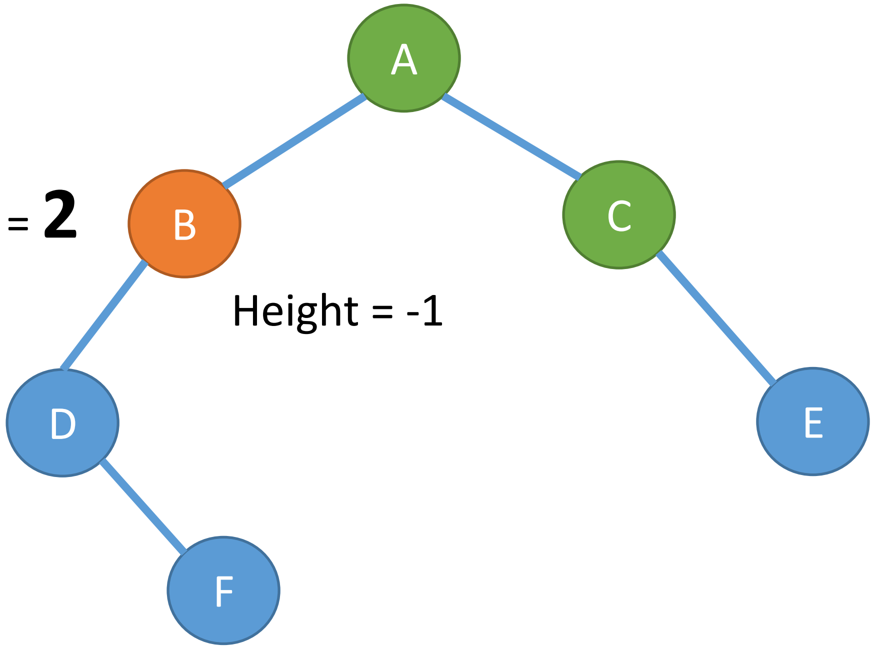
AVL Trees

NOT BALANCED

Balance factor = 1 – -1 = **2**

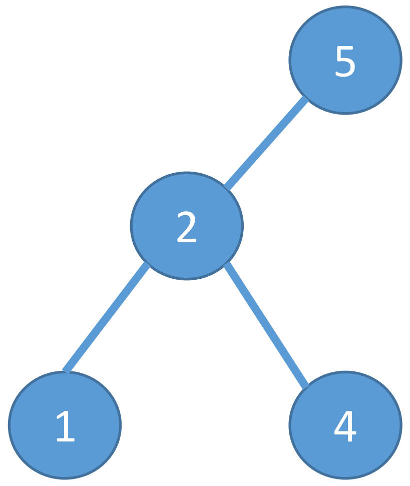
Height = -1

Height = 1



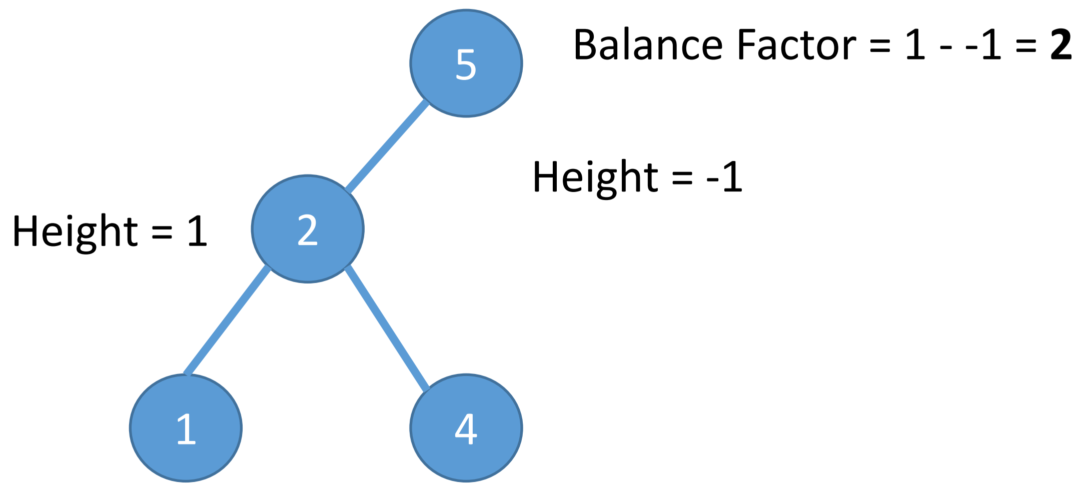
Rotation

- The self-balancing ability of the AVL tree is due to a rotation process.
- The rotation rearranges the nodes to balance the tree (or subtree).



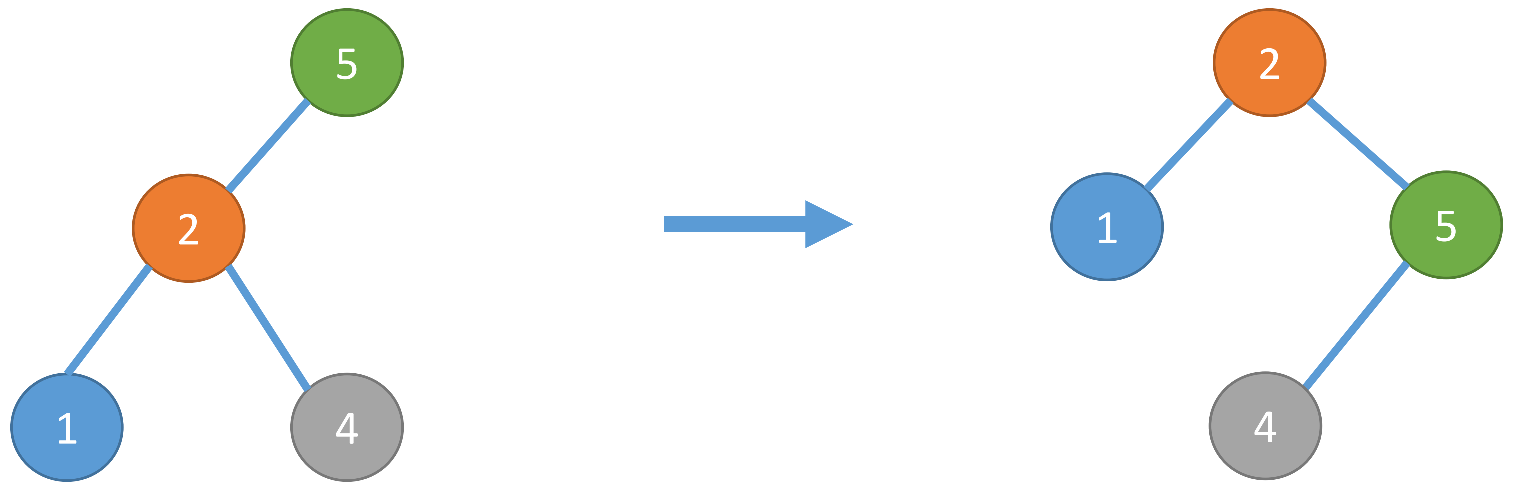
Rotation

- The root node needs to be rotated.
  - It is unbalanced on its left side, so it needs to be *right rotated*
  - **A balance factor > 1 indicates a right rotation is needed**



Right Rotation

- Right Rotation process
  - Node to rotate is demoted to its left child's right child
  - The node to rotate's (former) left child's right child becomes its left child

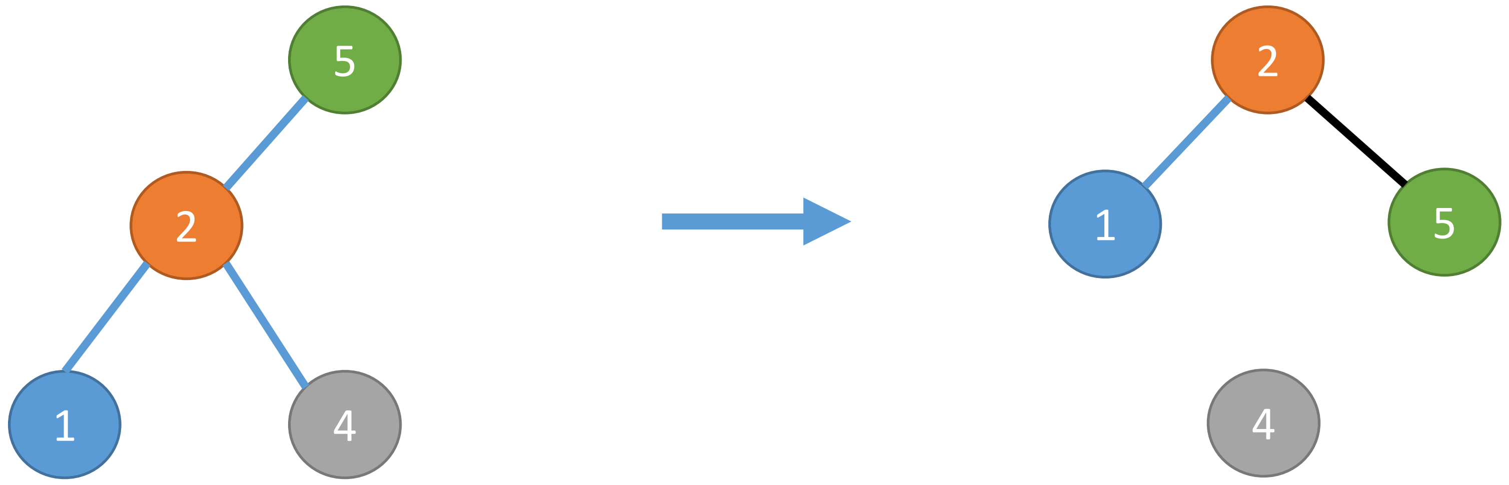


Unbalanced

Balanced

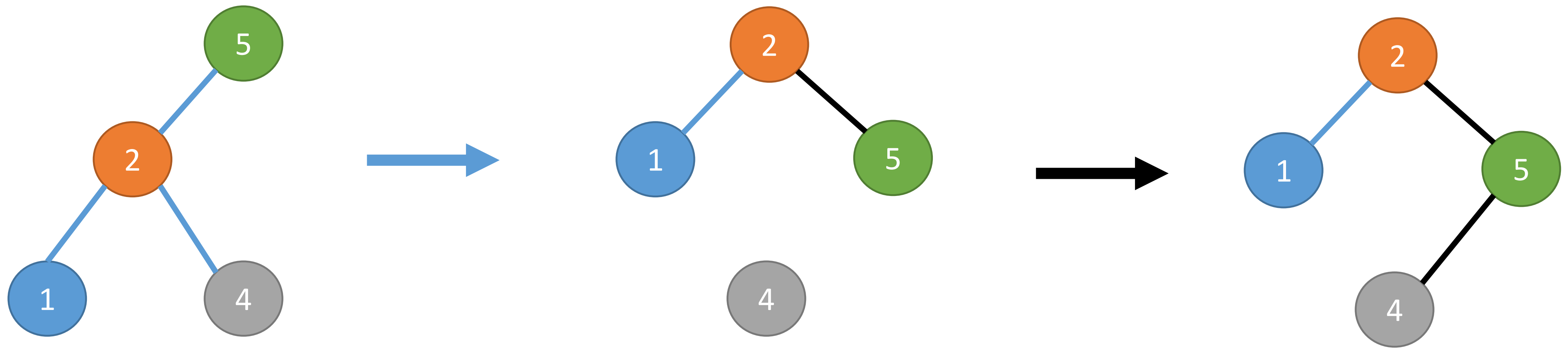
Right Rotation

- Right Rotation process
  - **Node to rotate (5) is demoted to its left child's (2) right child**
  - The node to rotate's (former) left child's right child becomes its left child



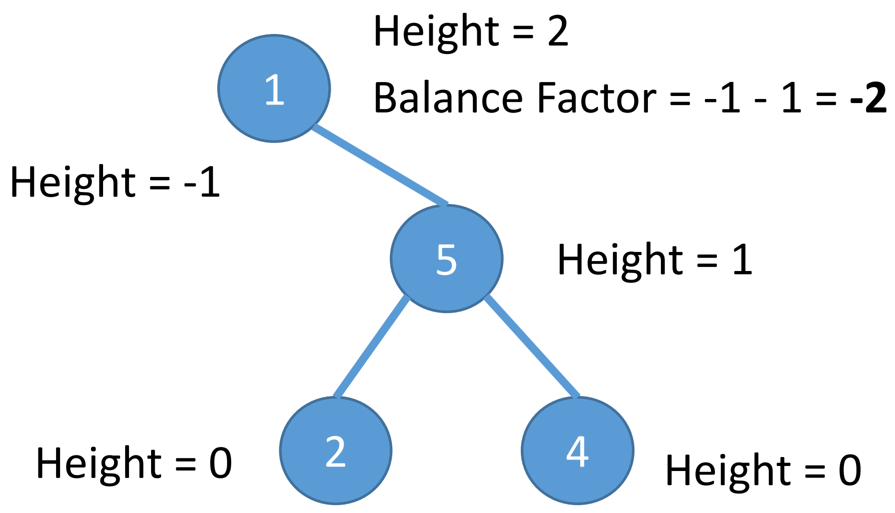
Right Rotation

- Right Rotation process
  - Node to rotate is demoted to its left child's right child
  - **The node to rotate's (5) (former) left child's (2) right child (4) becomes its left child**



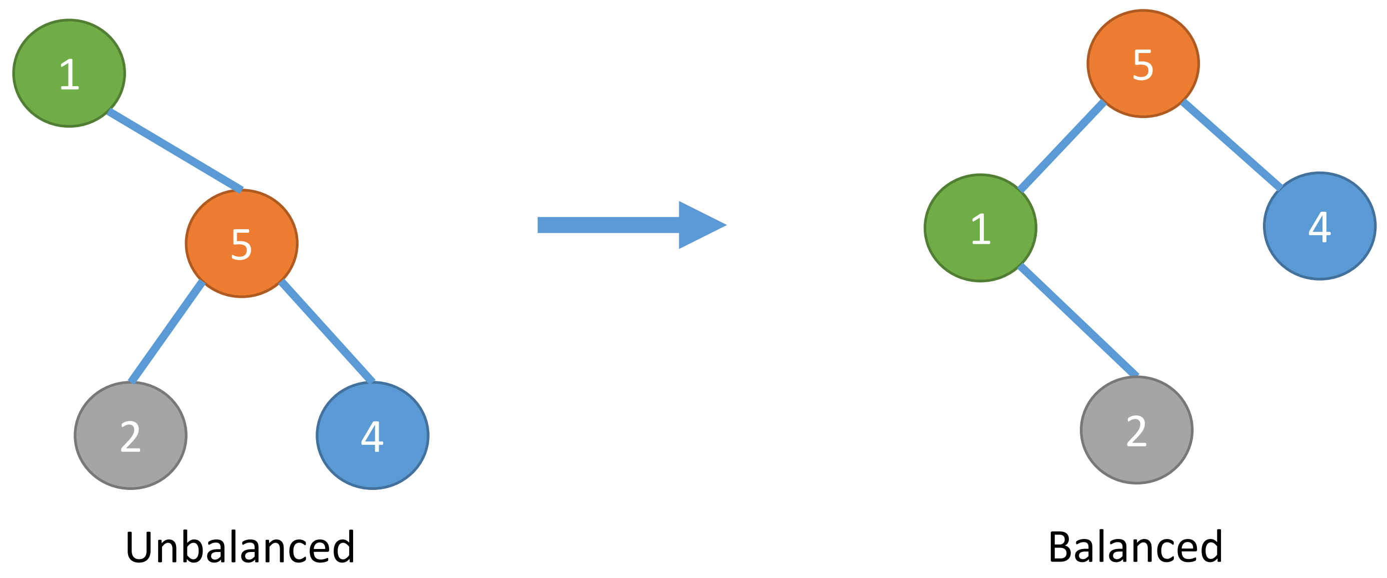
Rotation

- The root node needs to be rotated.
  - It is unbalanced on the right side, so it needs to be *left rotated*
  - **A balance factor < -1 indicates a left rotation is needed**



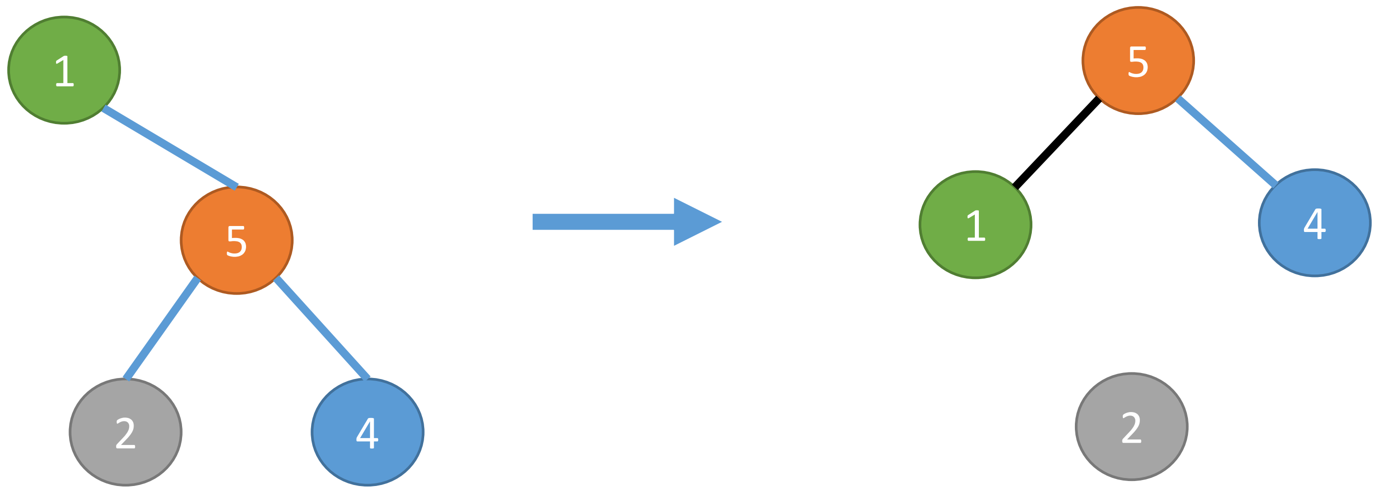
Left Rotation

- Left Rotation process
  - Node to rotate is demoted to its right child's left child
  - The node to rotate's (former) right child's left child becomes its right child



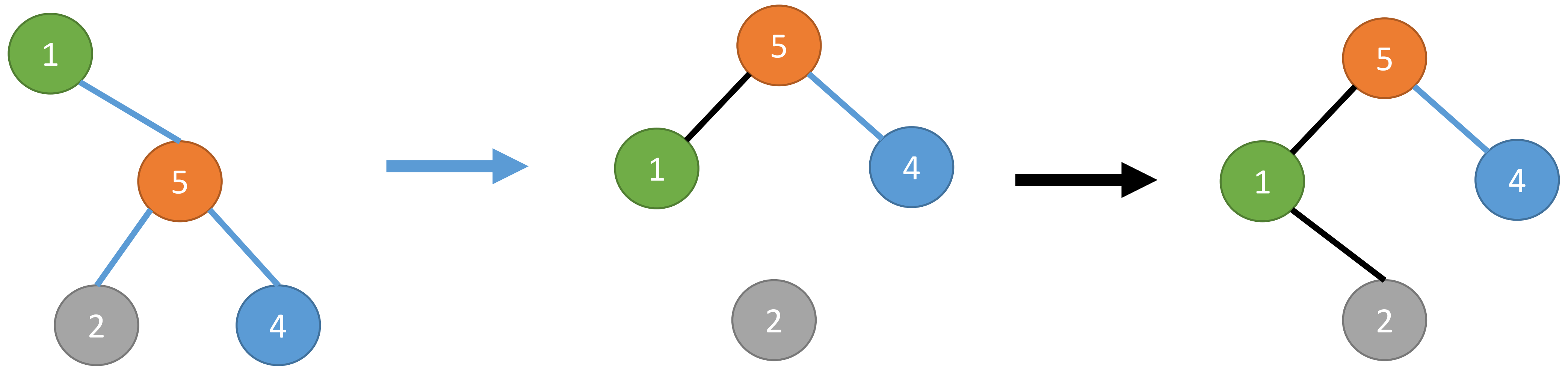
Left Rotation

- Left Rotation process
  - **Node to rotate (1) is demoted to its right child's (5) left child**
  - The node to rotate's (former) right child's left child becomes its right child



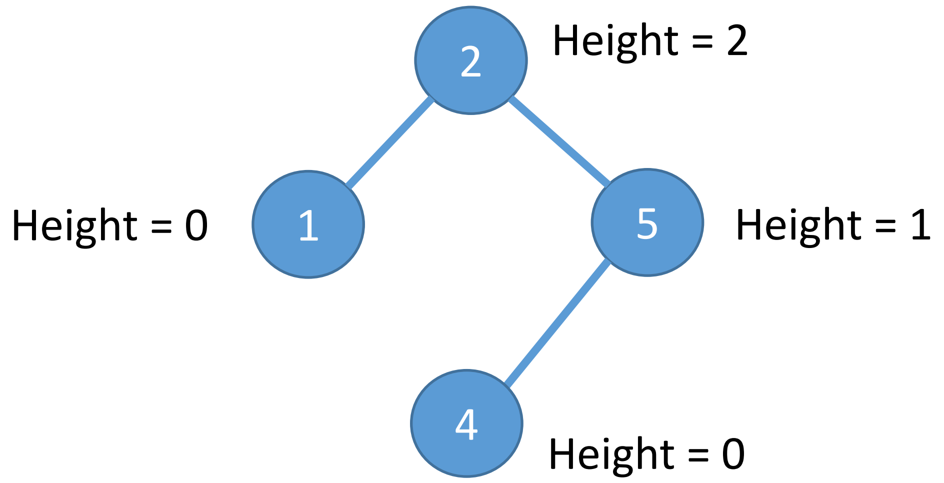
Left Rotation

- Left Rotation process
  - Node to rotate is demoted to its right child's left child
  - **The node to rotate's (1) (former) right child's (5) left child (2) becomes its right child**



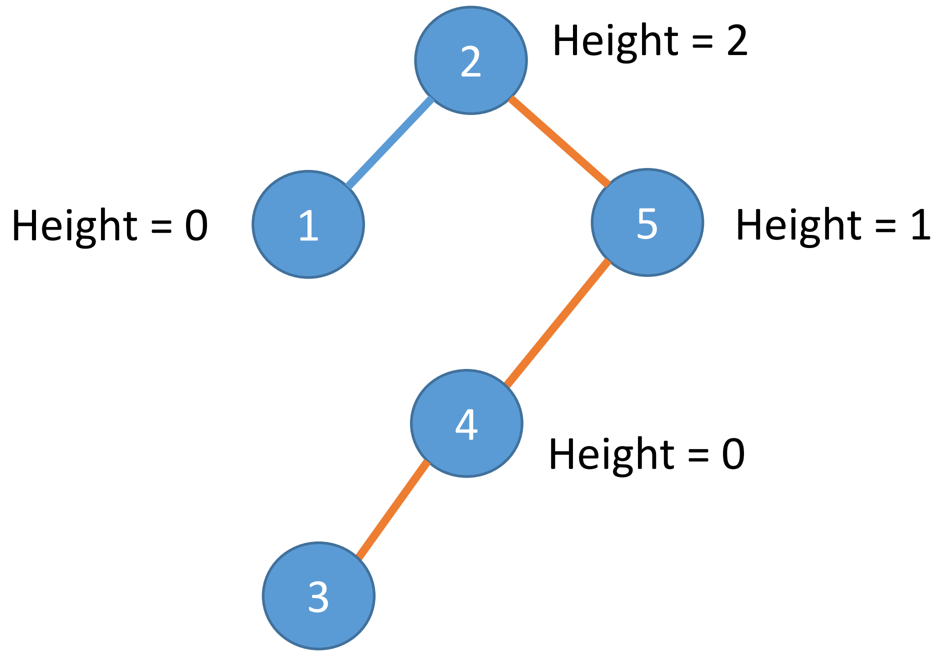
Insertion

- We want to add 3 to the AVL tree below



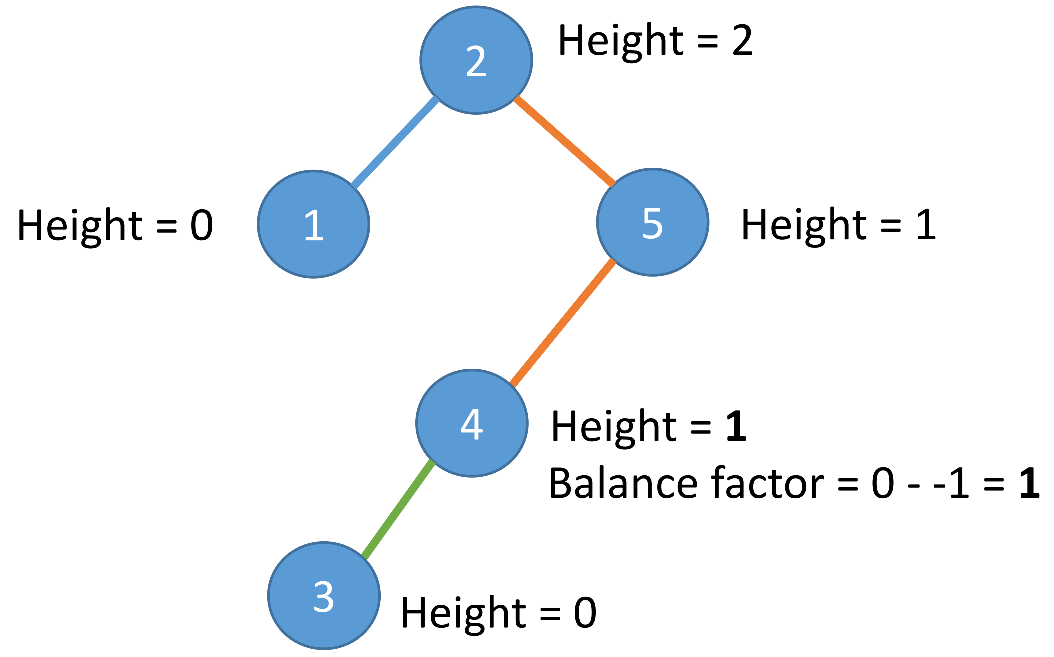
Insertion

- Traverse where 3 belongs and insert it
  - Just as you would do for a normal BST



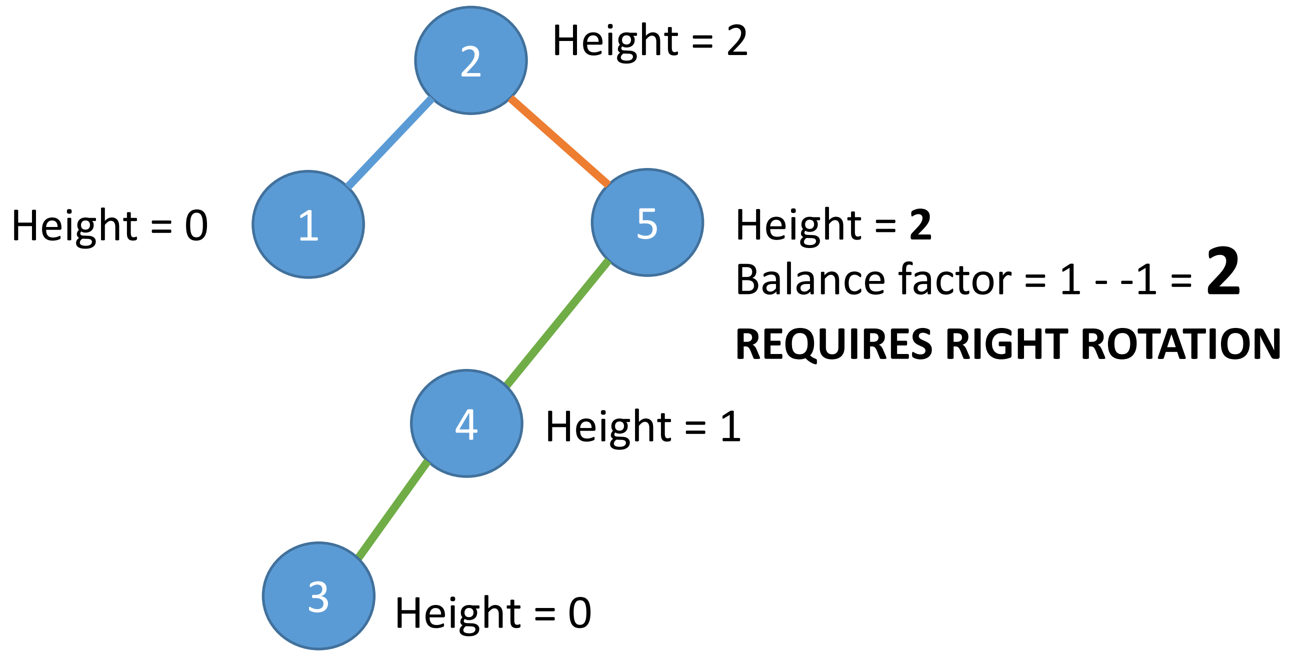
Insertion

- Work our way back up, adding 1 to each nodes height (if necessary), and checking if any rotation is necessary



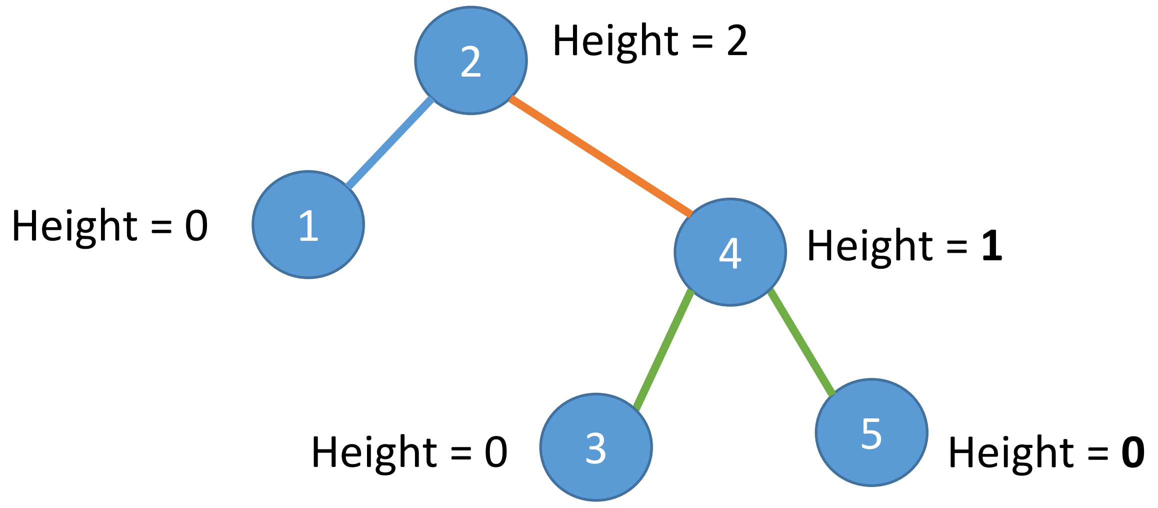
Insertion

- Work our way back up, adding 1 to each nodes height (if necessary), and checking if any rotation is necessary



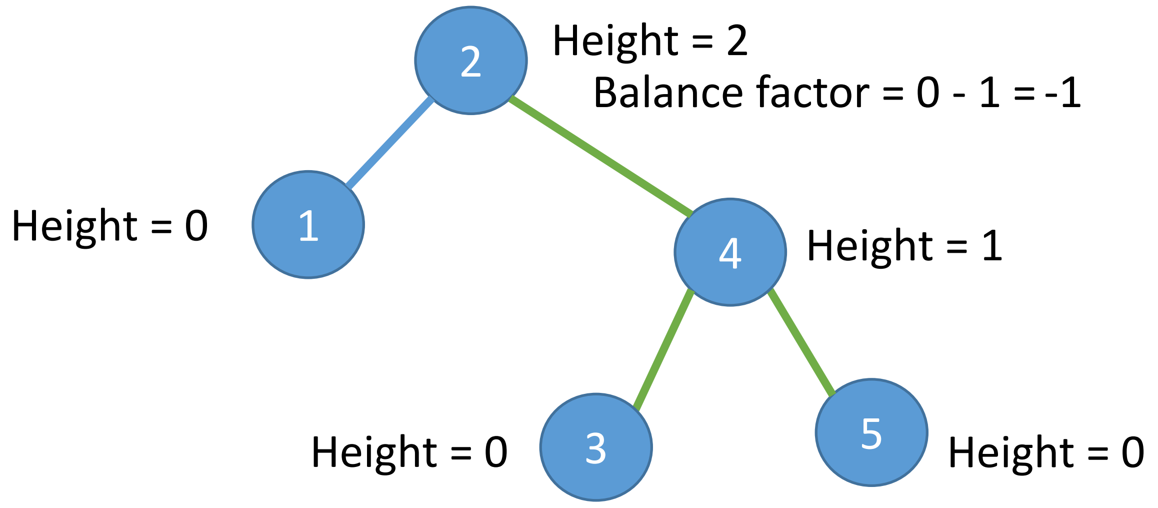
Insertion

- Work our way back up, adding 1 to each nodes height (if necessary), and checking if any rotation is necessary



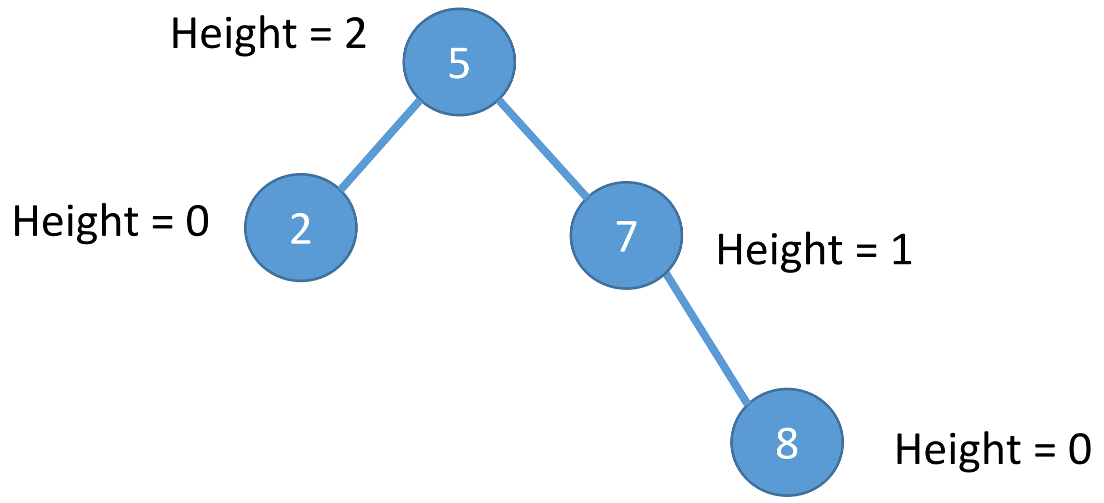
Insertion

- Work our way back up, adding 1 to each nodes height (if necessary), and checking if any rotation is necessary



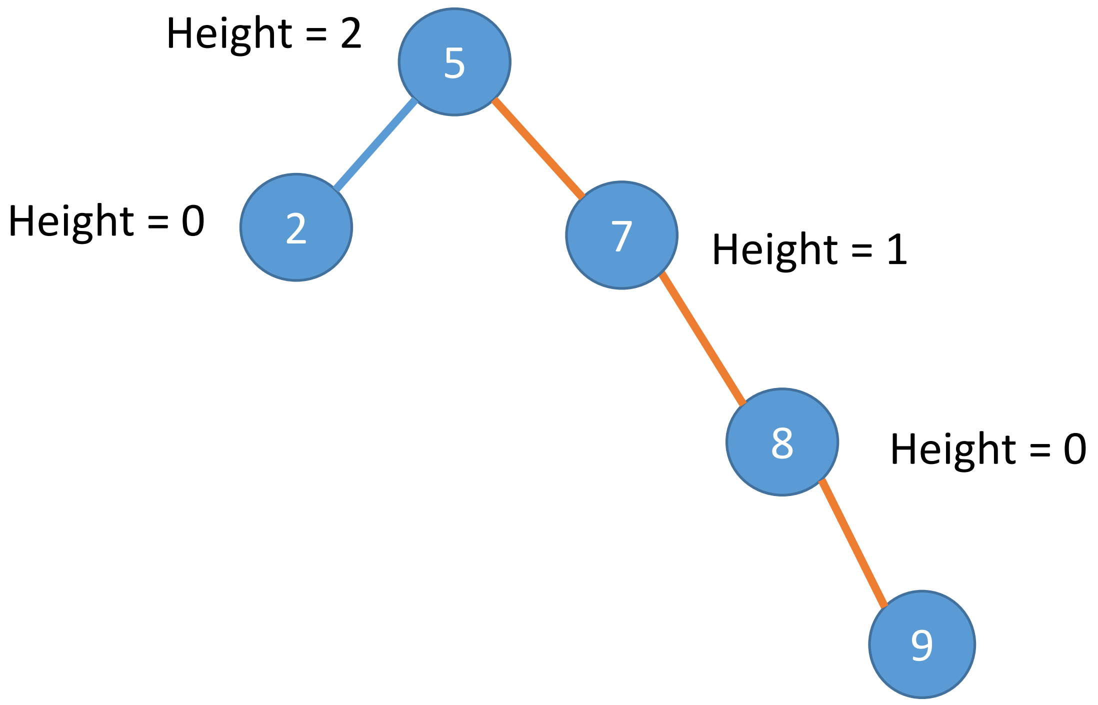
Insertion

- We want to add 9 to the AVL tree below



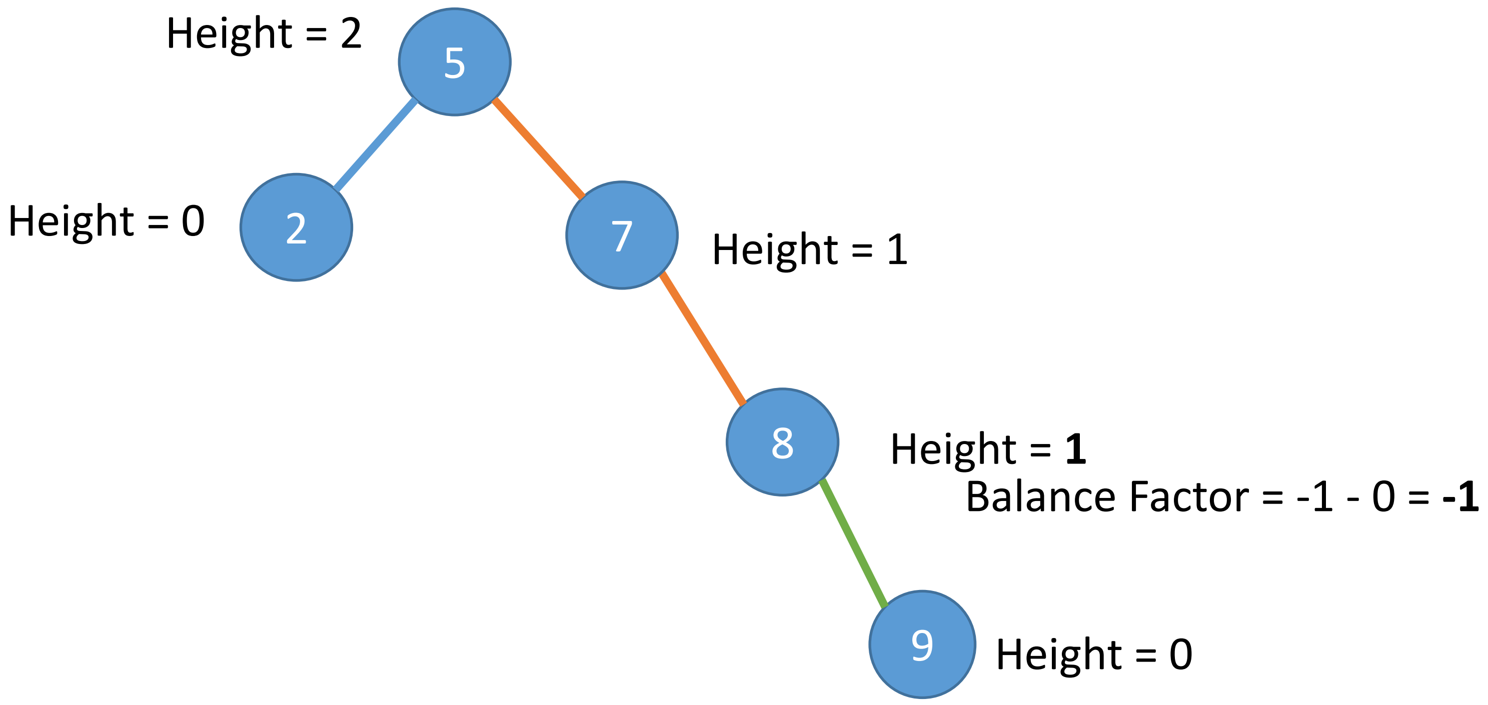
Insertion

- Traverse where 9 belongs and insert it
  - Just as you would do for a normal BST



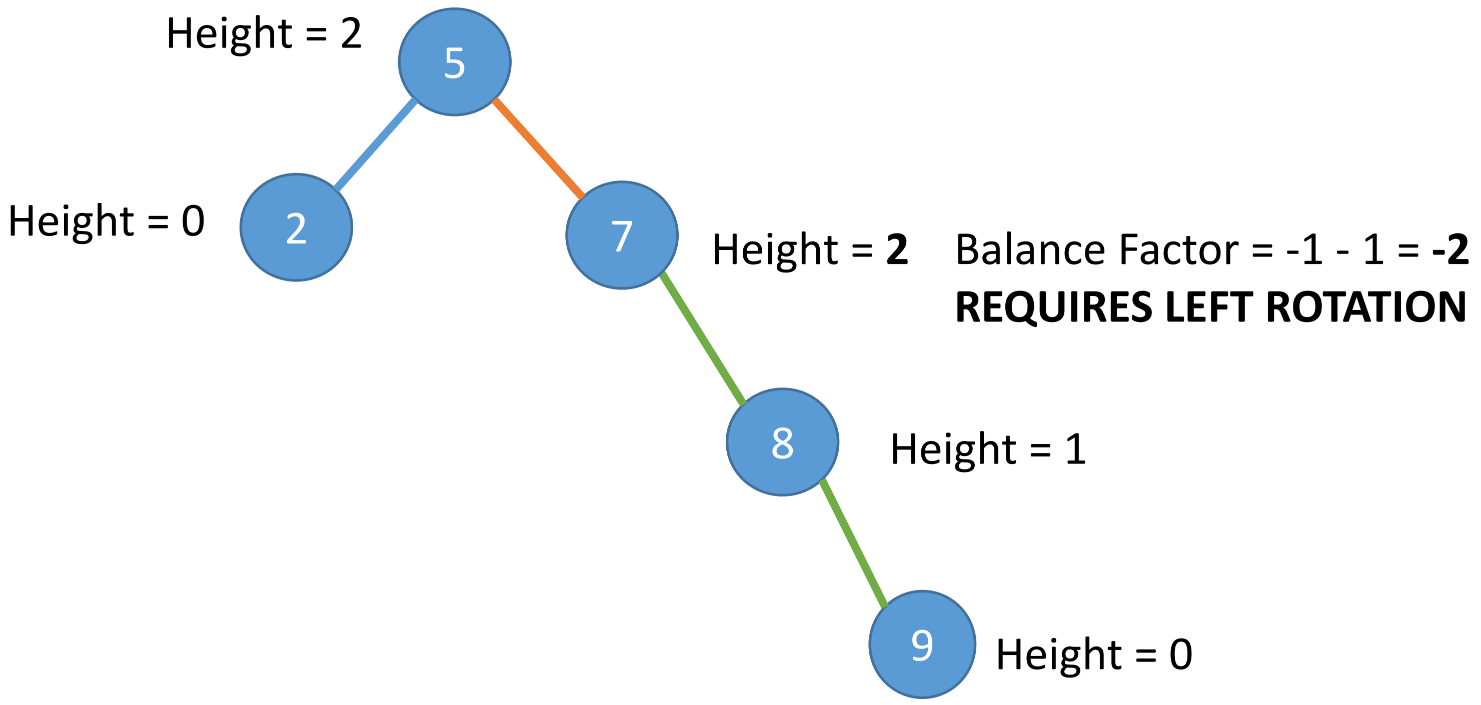
Insertion

- Work our way back up, adding 1 to each nodes height (if necessary), and checking if any rotation is necessary



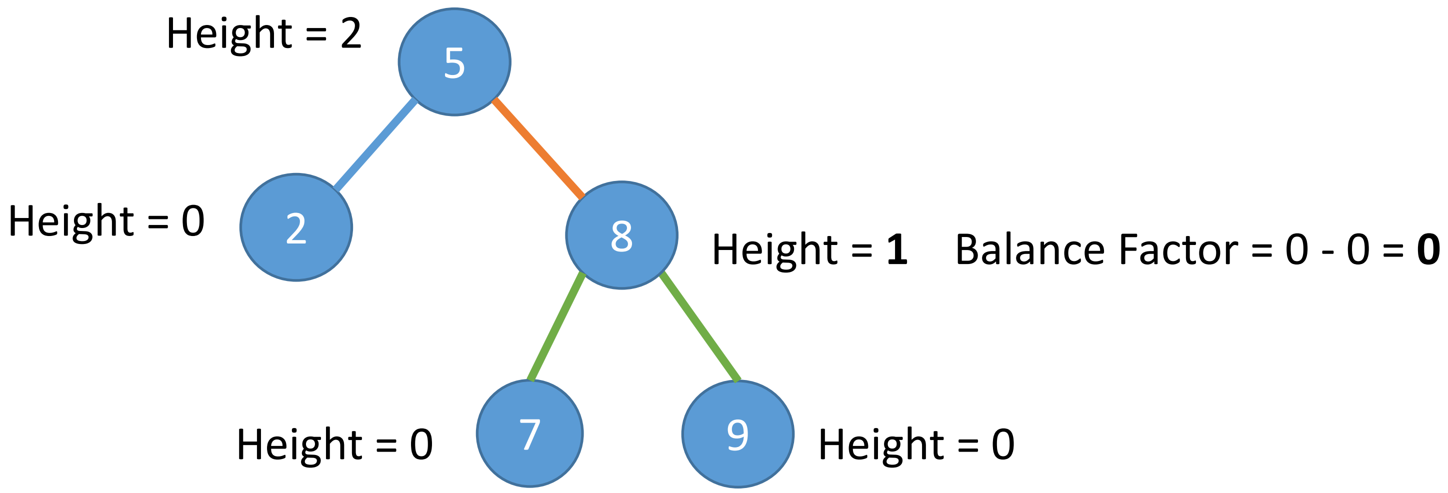
Insertion

- Work our way back up, adding 1 to each nodes height (if necessary), and checking if any rotation is necessary



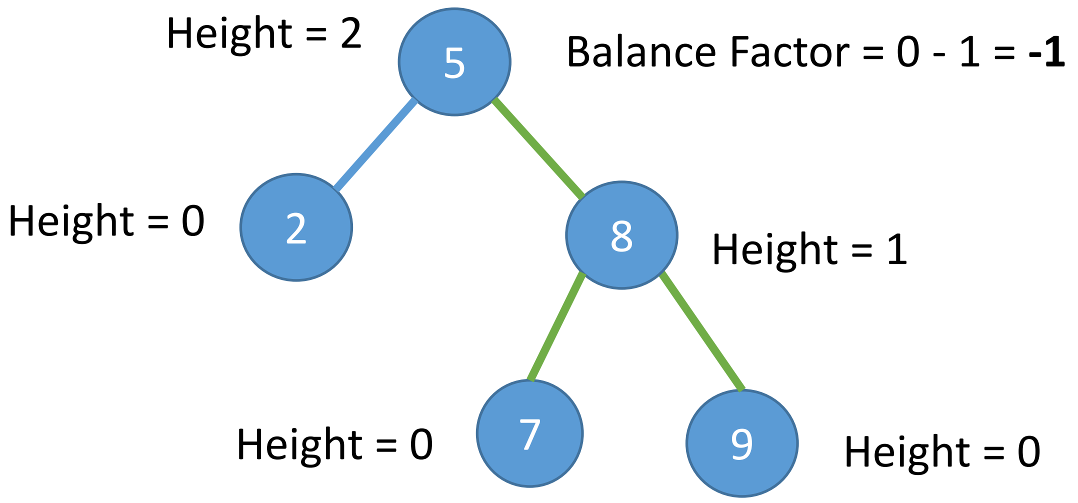
Insertion

- Work our way back up, adding 1 to each nodes height (if necessary), and checking if any rotation is necessary



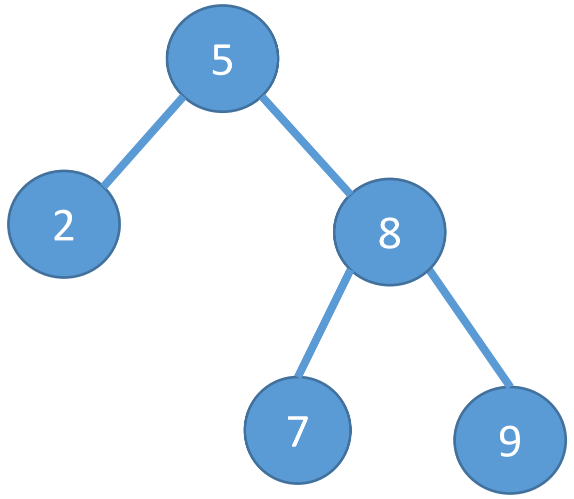
Insertion

- Work our way back up, adding 1 to each nodes height (if necessary), and checking if any rotation is necessary



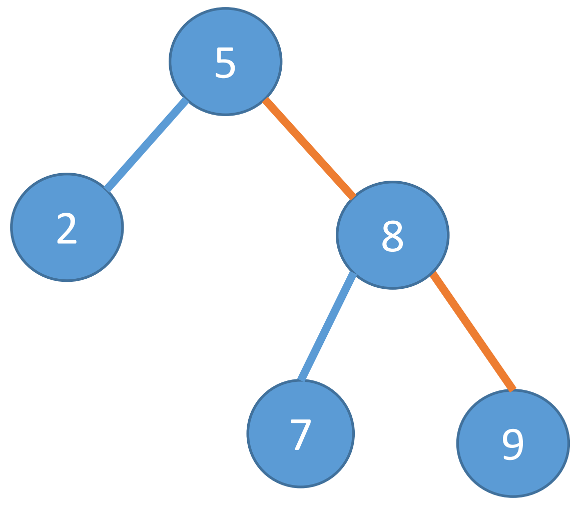
Removal

- Removing a node is similar to the process for removal from a normal BST.
  - After removing a node, a rotation might be needed somewhere up the tree



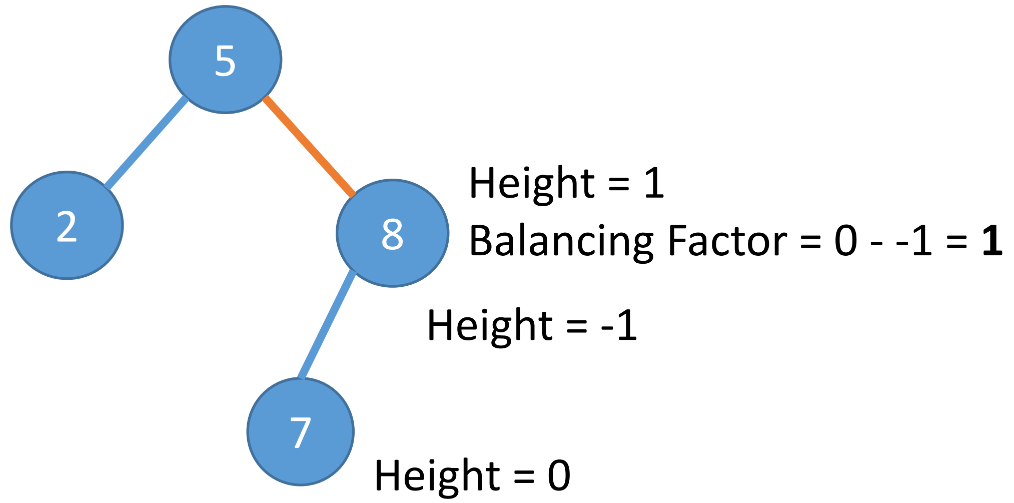
Removal

- Removing 9



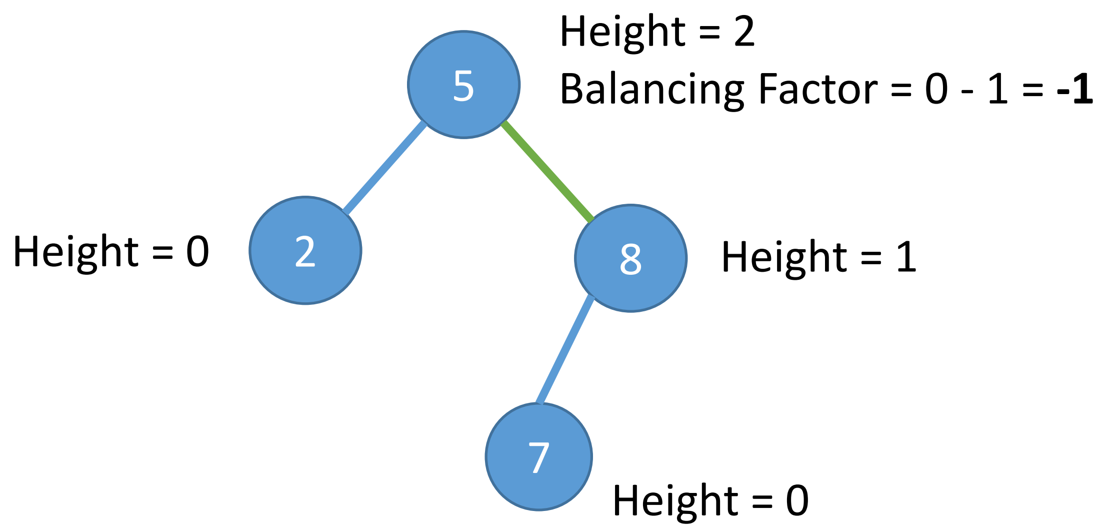
Removal

- Working our way back up to the root node…



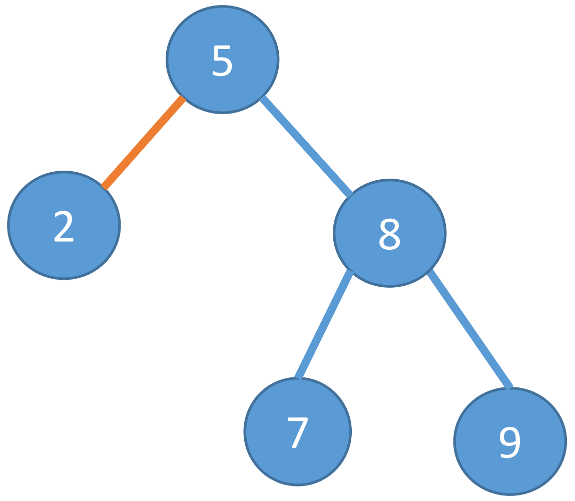
Removal

- No rotation was necessary.



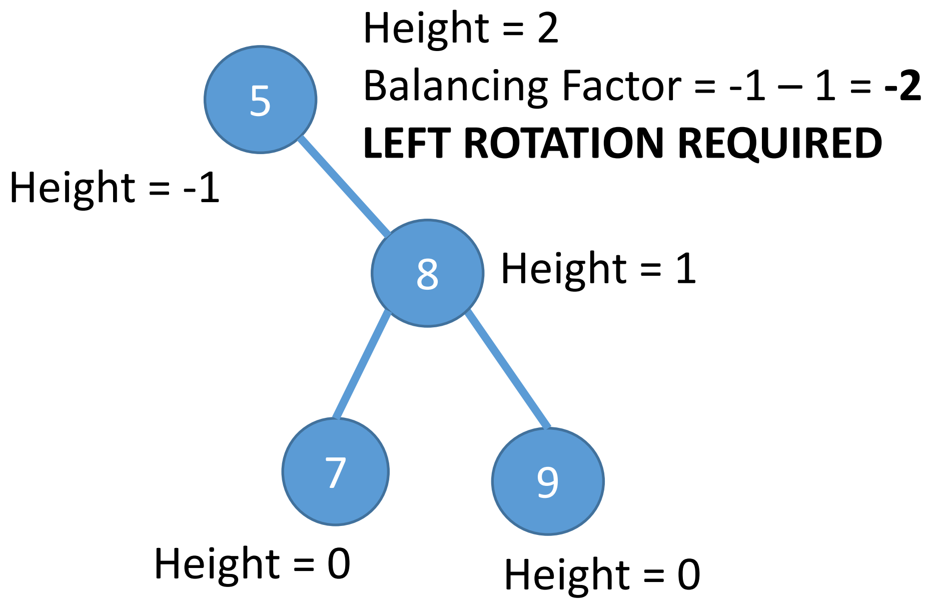
Removal

- Removing 2

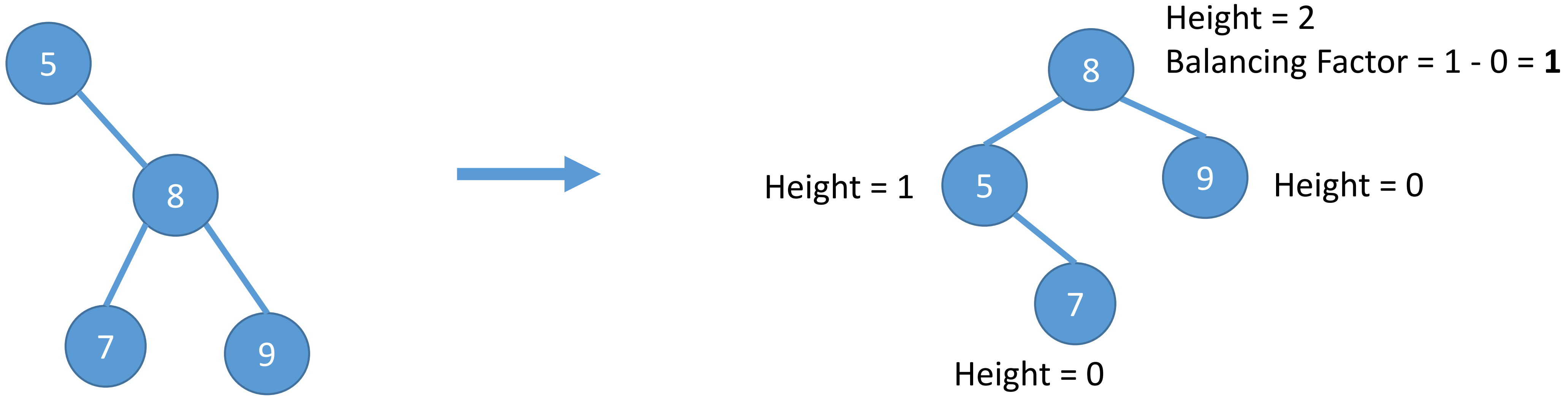


Removal

- At the root node…


Height = 2
Balancing Factor = -1 – 1 = **-2**
LEFT ROTATION REQUIRED
5
Height = -1
8
Height = 1
7
9
Height = 0
Height = 0

Removal


5
8
7
9
Height = 2
Balancing Factor = 1 - 0 = **1**
8
Height = 1
5
9
Height = 0
7
Height = 0

Red-Black Trees

- A **Red-Black tree** is a binary search tree that also self-balances.

- Nodes in the tree are either “red” or “black”
  - Node can contain other data/information, but each will be either “red” or “black”
  - Red = 0
  - Black = 1

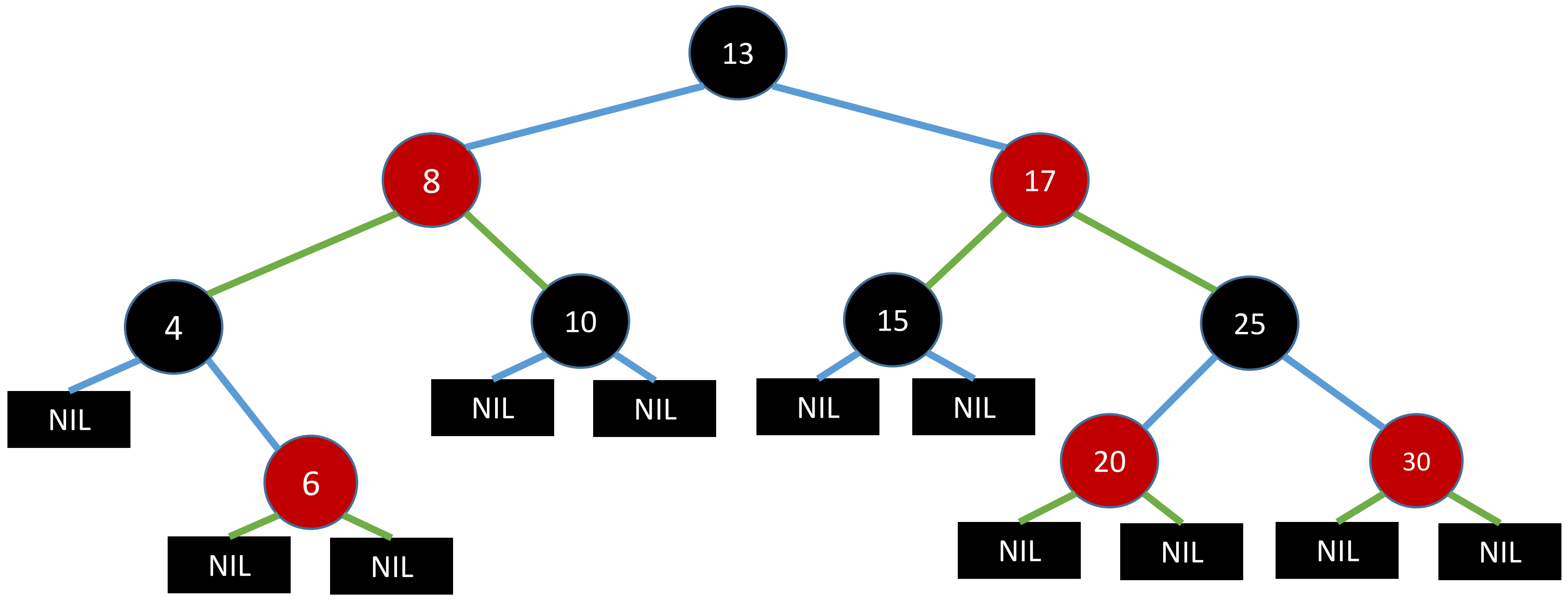
Red-Black Trees

- Important Properties
  1. A red node cannot be adjacent to another red node
     - In other words, a red node cannot have red children
  2. The path from any node to any leaf always contains the same number of black nodes
     - In other words, the sum of the colors from any node to any leaf is the same
  3. The path from the root to the farthest leaf is never more than twice as long as the root to the nearest leaf

- Simplification Tricks
  - The root node can always be black
    - Red-Black Trees are usually implemented this way
  - An absent (nil) child can always be treated as a black node
    - Ensures every real node has two children with a defined color
    - Meaning a leaf in a Red-Black tree does not contain any data
      - A null child pointer indicates a black leaf

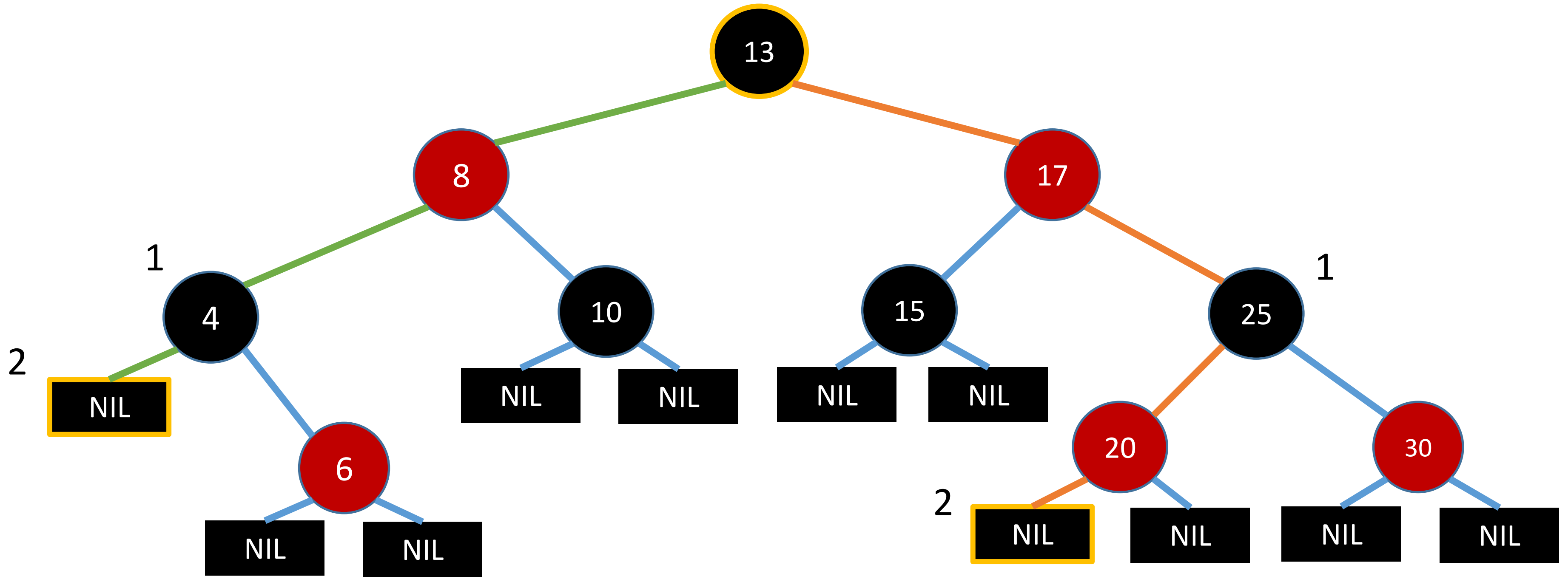
Red-Black Trees

1. A red node cannot be adjacent to another red node



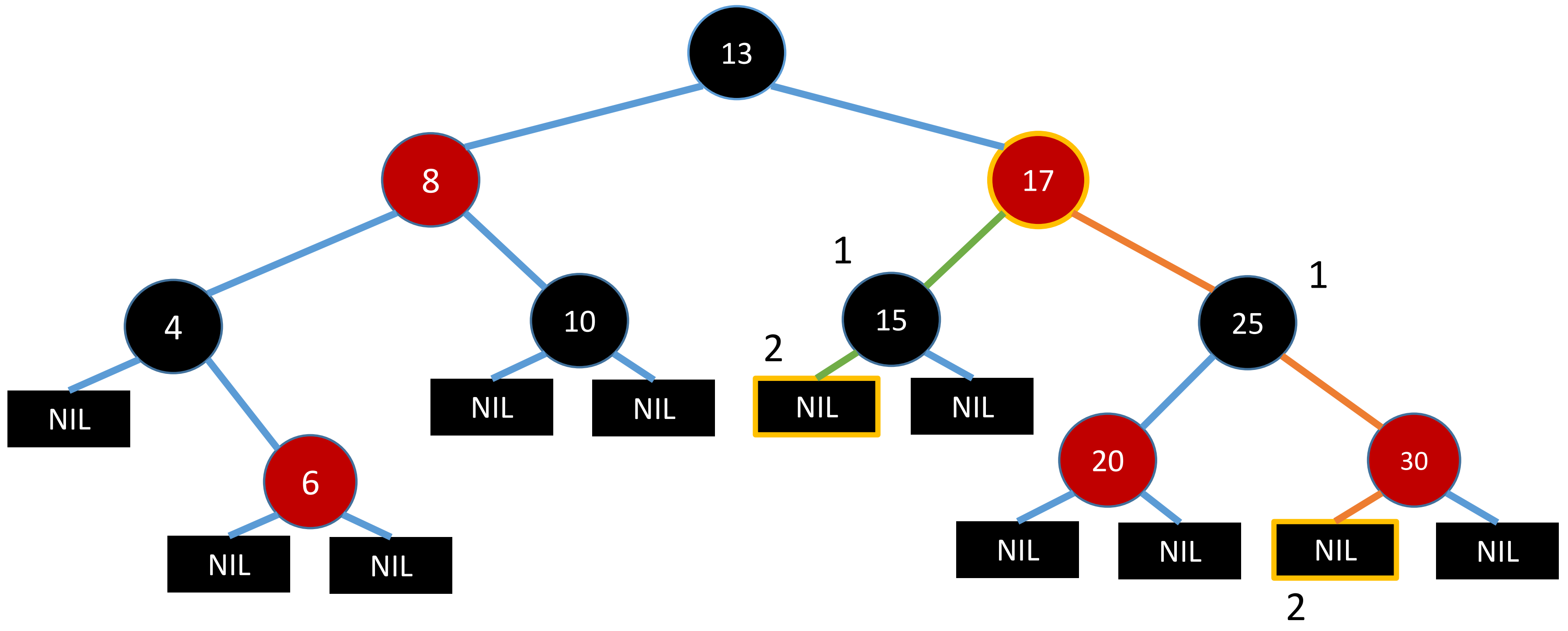
Red-Black Trees

2. The path from any node to any leaf always contains the same number of black nodes.



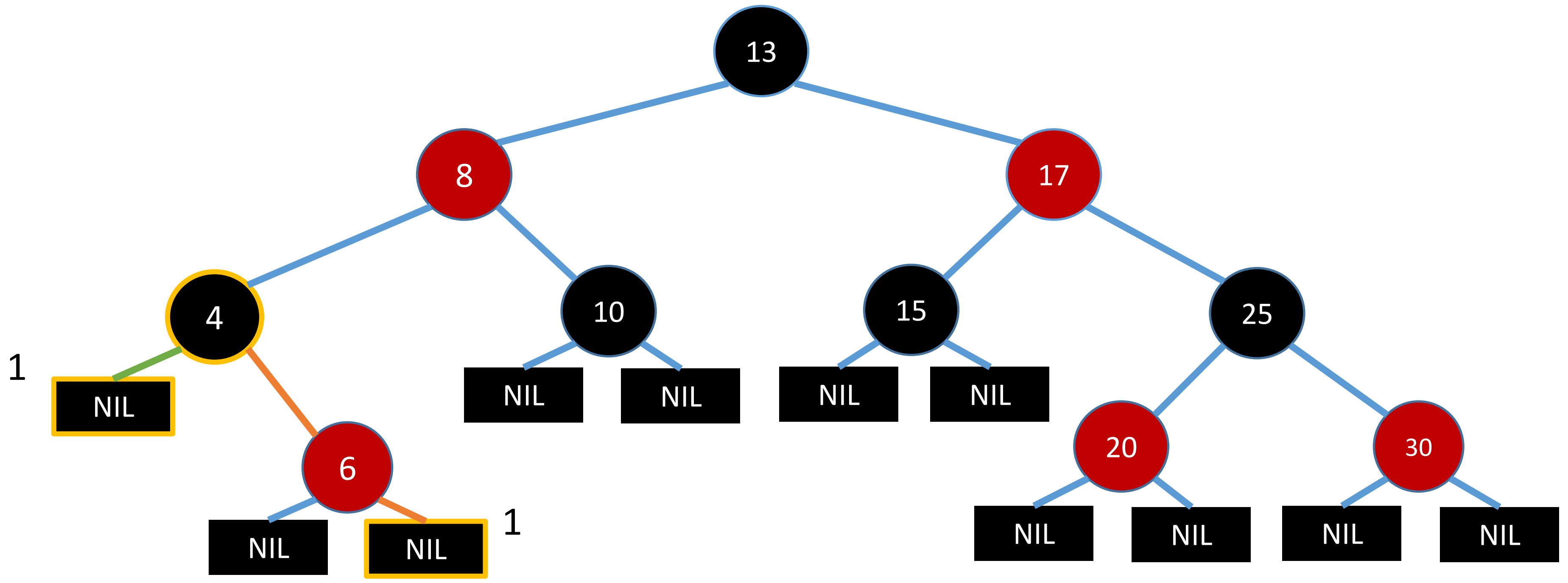
Red-Black Trees

2. The path from any node to any leaf always contains the same number of black nodes.



Red-Black Trees

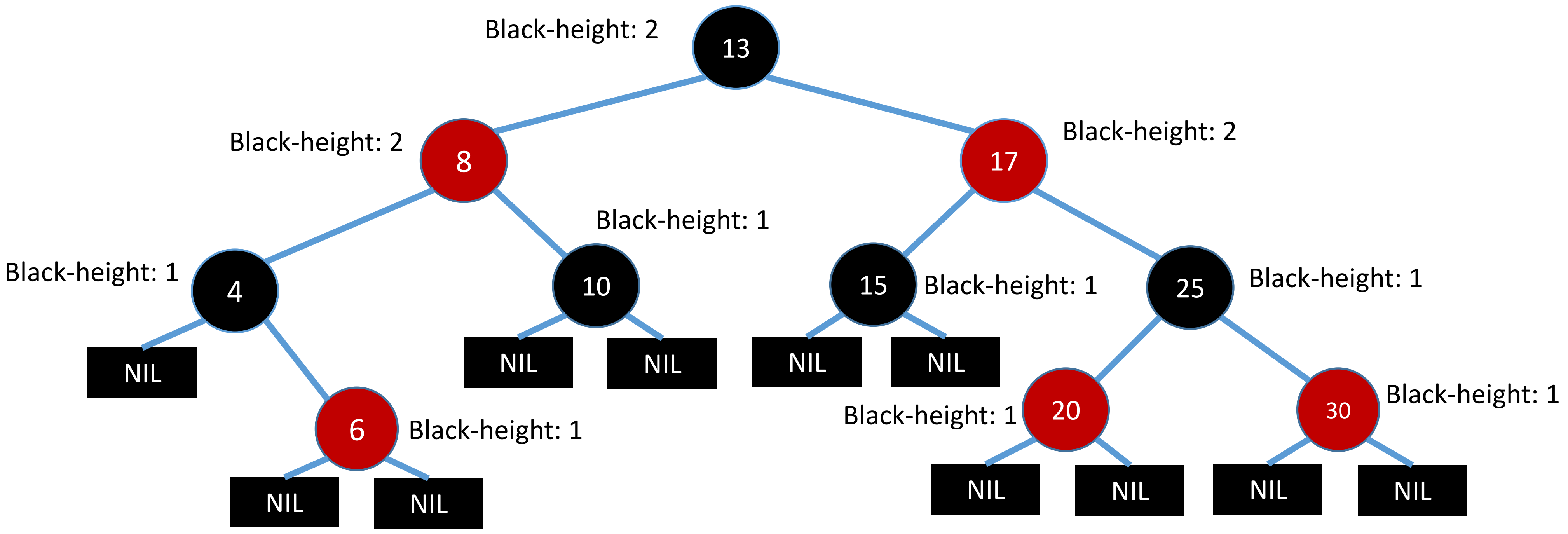
2. The path from any node to any leaf always contains the same number of black nodes.



Red-Black Trees

- This path is also known as a node’s **black-height**
- As depicted on the previous slides, this path does not include the starting node (if it is black)

Red-Black Trees



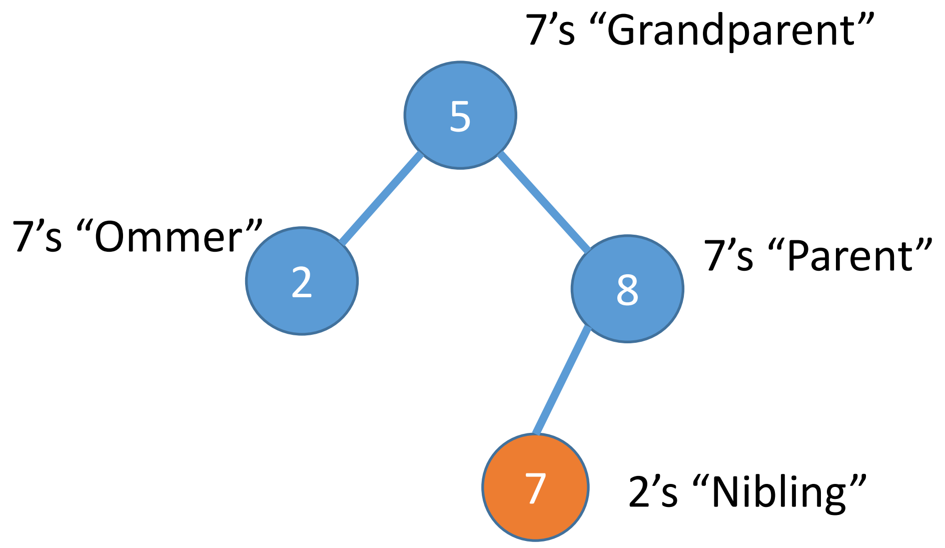
- The "black-height of the tree" is the black-height of the root node
    - In this case, the black-height of the tree is 2

Rotation

- When a node is added or removed, the tree may become unbalanced
- Like an AVL Tree, Red-Black Trees self-balance using rotation
- The rotation process in a Red-Black Tree is the same as the process used in an AVL Tree
    - However, the Red-Black Tree must be rotated/rebalanced so that its rules are met

Insertion

- We'll use some additional terminology when describing the insertion/rotation processes:
  - "Grandparent"- A node's parent's parent
  - "Ommer"- A node's parent's sibling
    - (Gender-neutral term for aunt/uncle)
  - "Nibling"- A node's sibling's child
    - (Gender-neutral term for niece/nephew)



Insertion

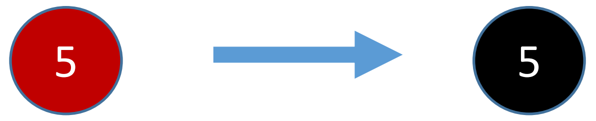
- There are two steps to inserting a new node:
    1. Find where the node belongs, insert it, and color it red
    2. Perform rotations and recoloring (if necessary)

- The first step is easy enough

- When to rotate and/or recolor depends on what scenario the violating node is in
    - There are 4 possible scenarios

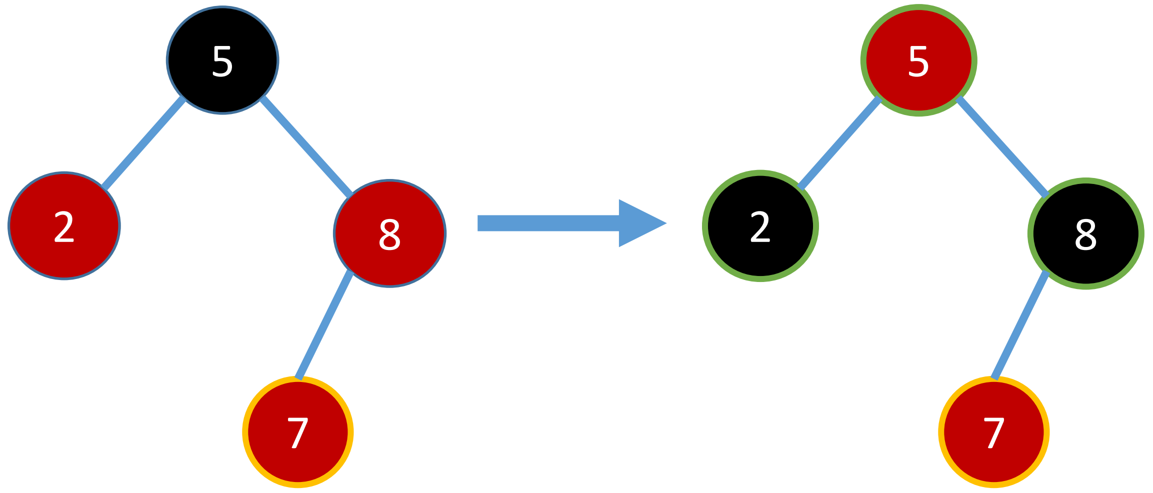
Insertion

- Scenario 1: The node is the root node
    - In other words, the tree was empty
- Insertion Process (Inserting 5):
    - The new node is inserted as the root and is colored red
    - Since it is the root, its color is simply changed to black



Insertion

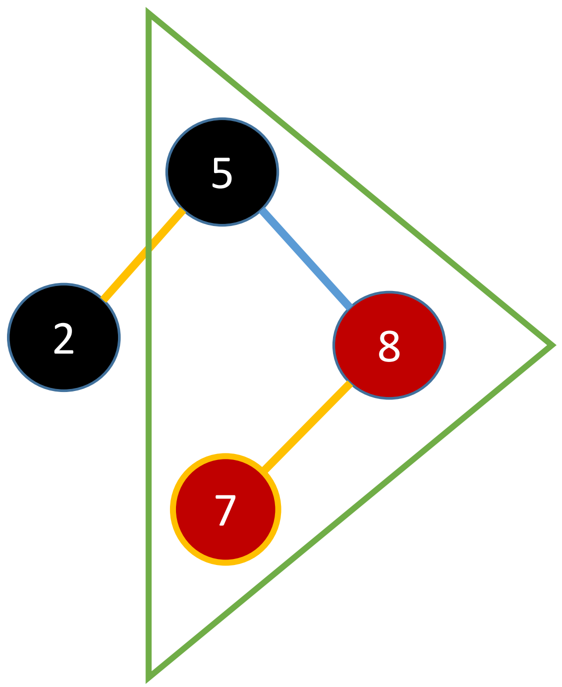
- Scenario 2: The node's ommer is red
- Recoloring Process:
  - In this scenario, the node's (7) parent, grandparent and ommer are recolored
    - There is no rotation in this scenario

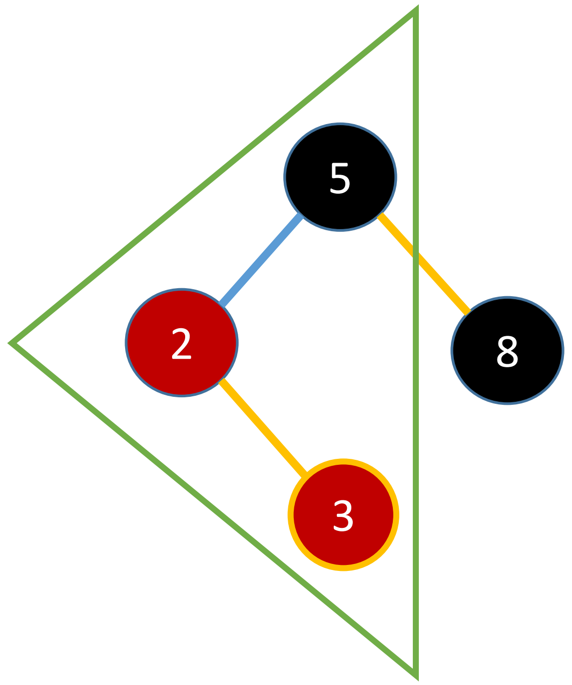


Note: 5 would eventually be recolored as we work our way up the tree checking for violations

Insertion

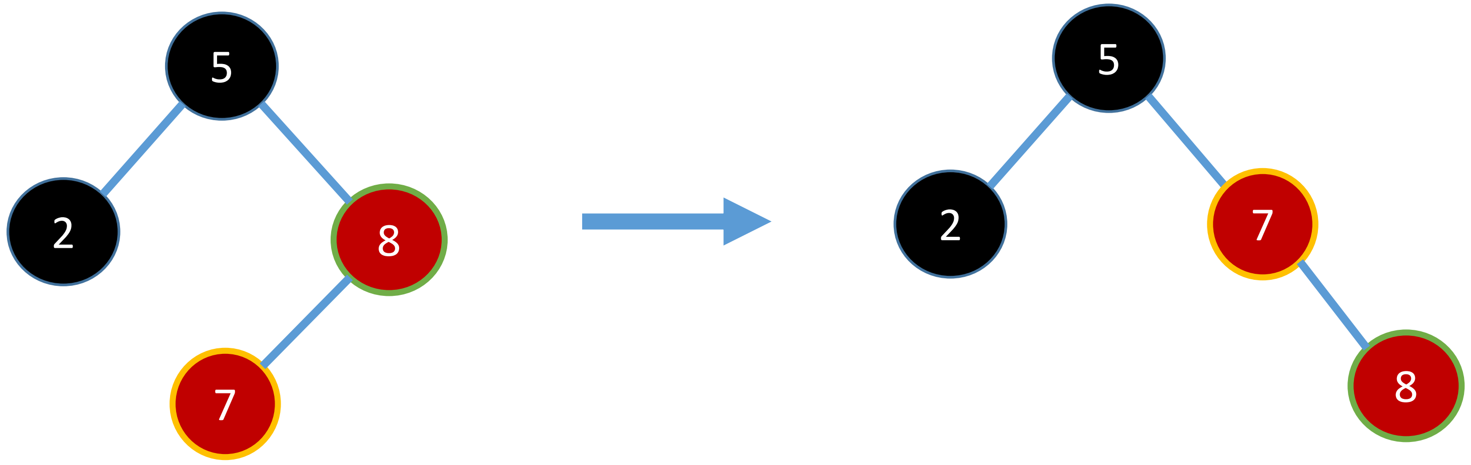
- Scenario 3: The node’s ommer is black and the subtree forms a *triangle*
  - In other words, the node and its *ommer* are **both** left (or **both** right) children





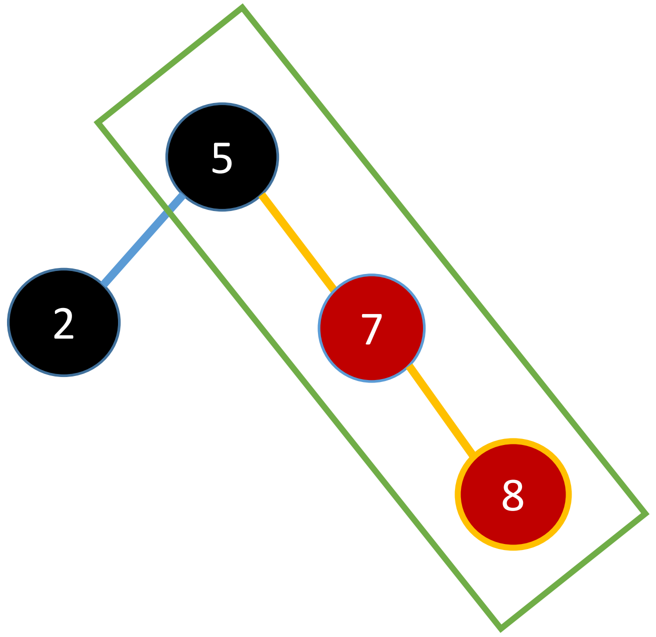
Insertion

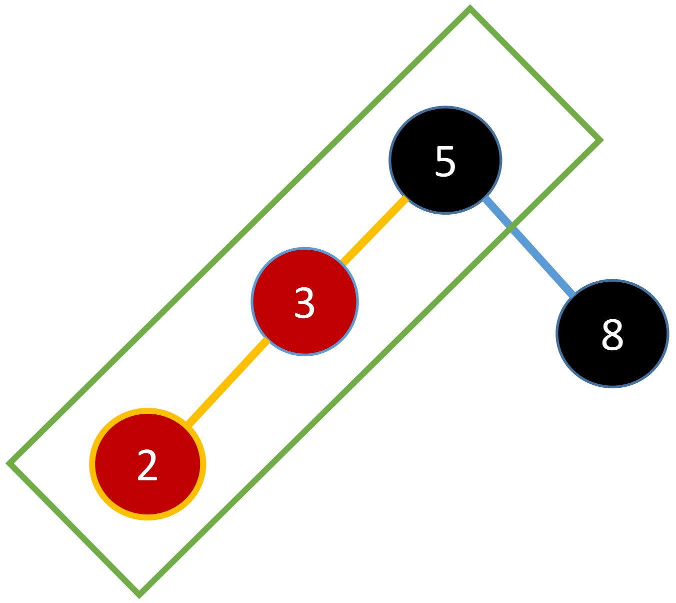
- Rotation Process:
  - In this scenario, the node’s parent is rotated in the opposite direction of the new node
  - Notice this does not necessarily fix the violation, but it leads us to the fourth and final scenario…



Insertion

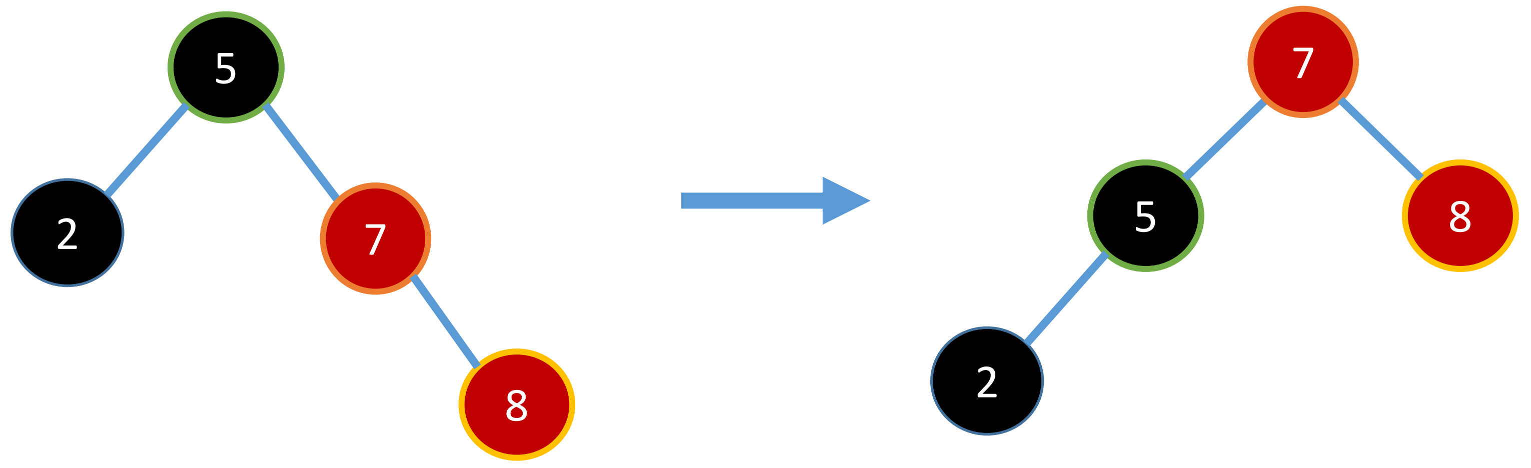
- Scenario 4: The node's ommer is black and the subtree forms a *line*
  - In other words, the node and its *parent* are **both** left (or **both** right) children





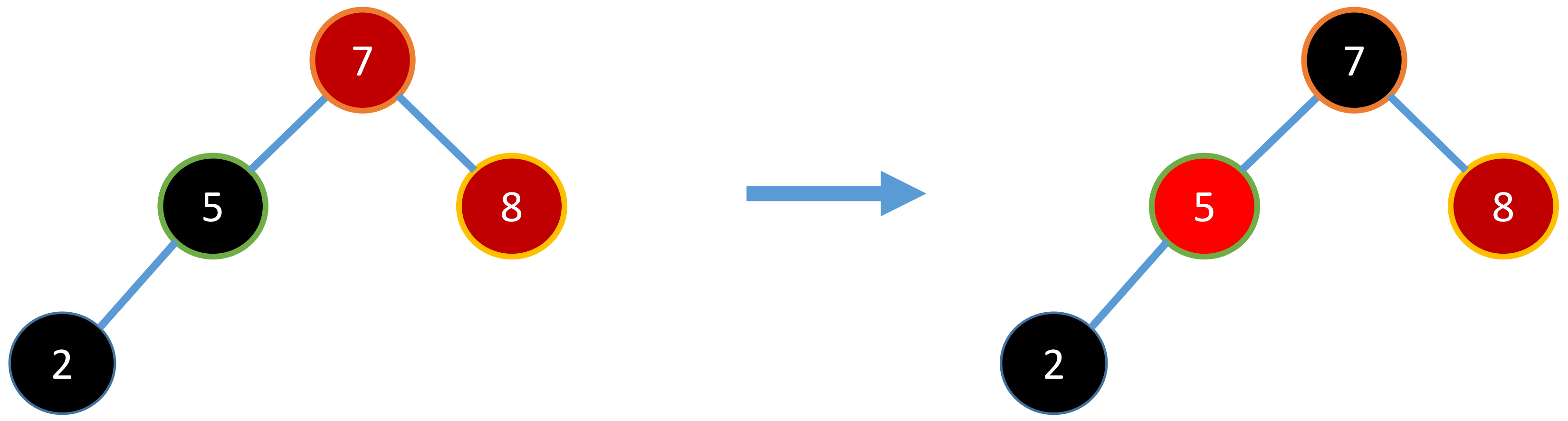
Insertion

- Rotation Process:
  - In this scenario, the node's grandparent is rotated in the opposite direction of the node



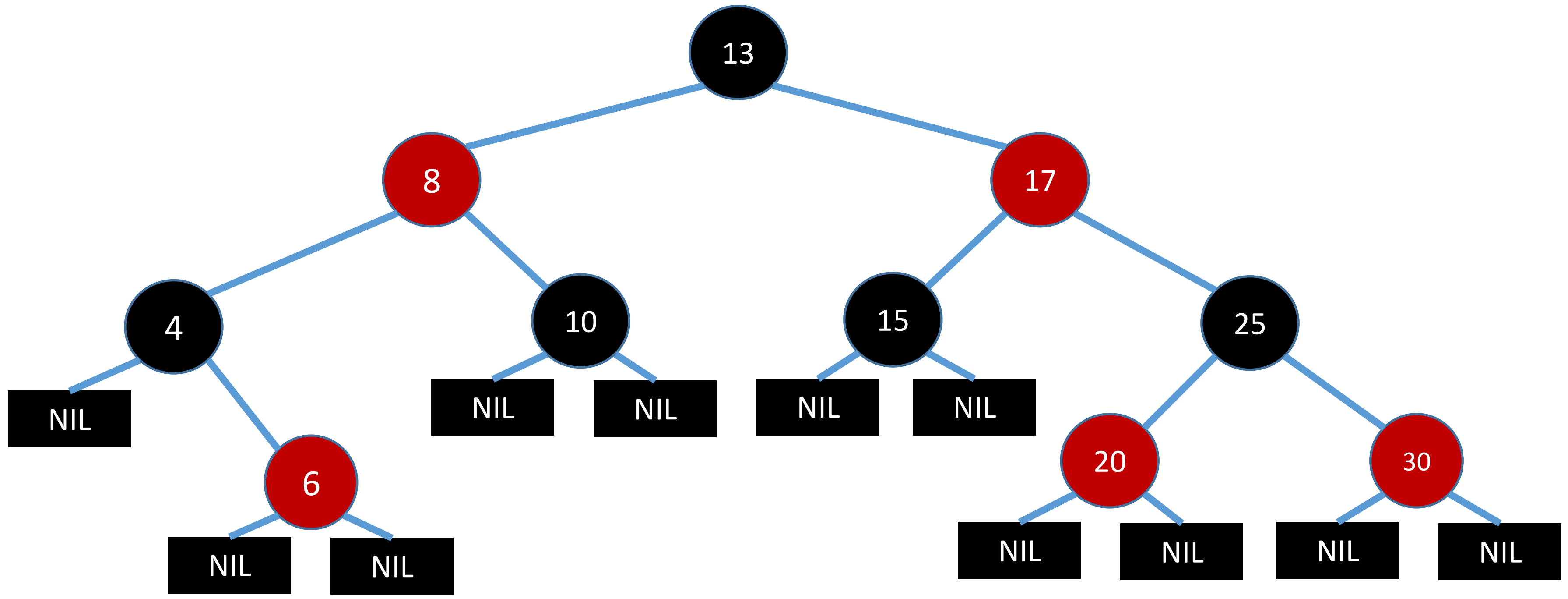
Insertion

- Recoloring Process:
  - Following the rotation, the node's original parent and grandparent are recolored



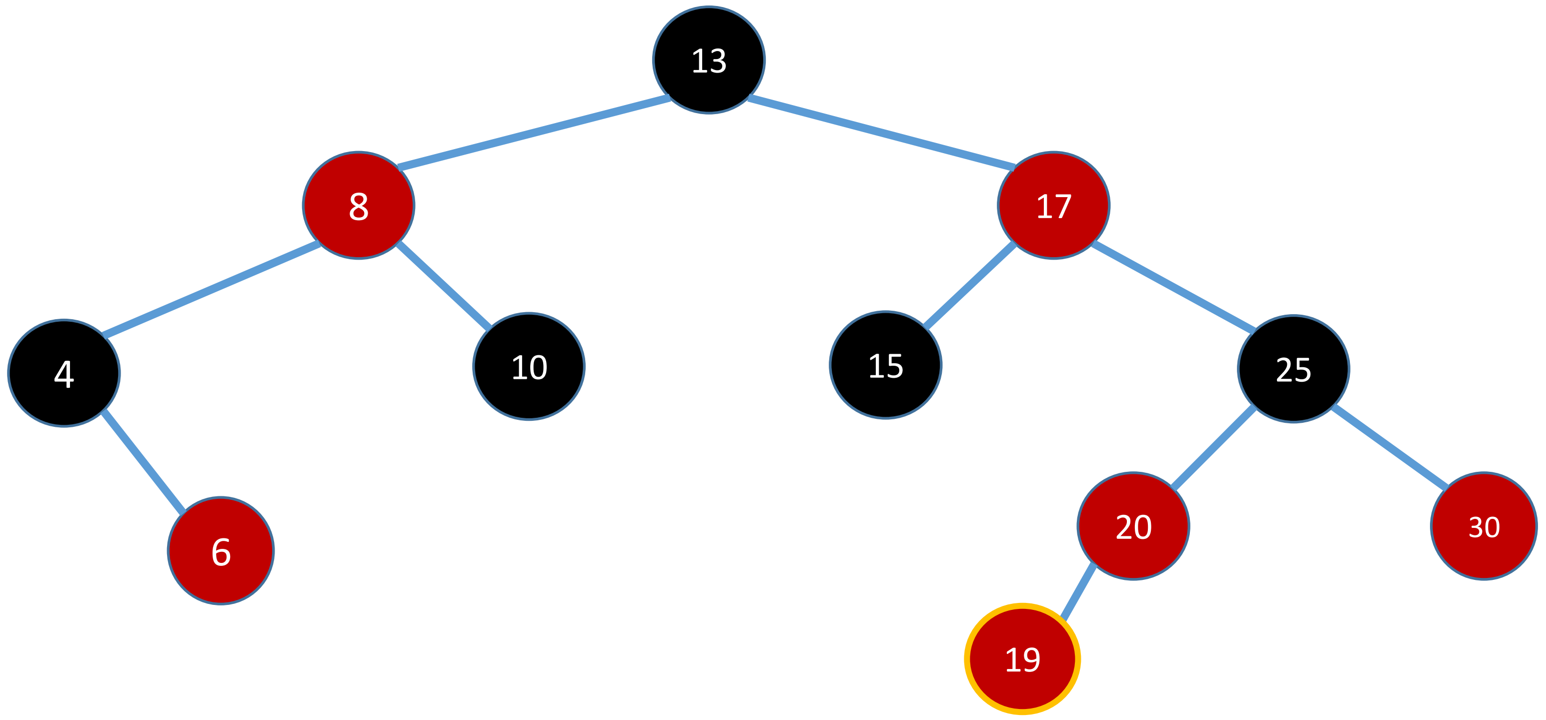
Red-Black Trees

- To demonstrate, we'll insert 19
  - For simplicity and less clutter, the NIL leaves will be removed



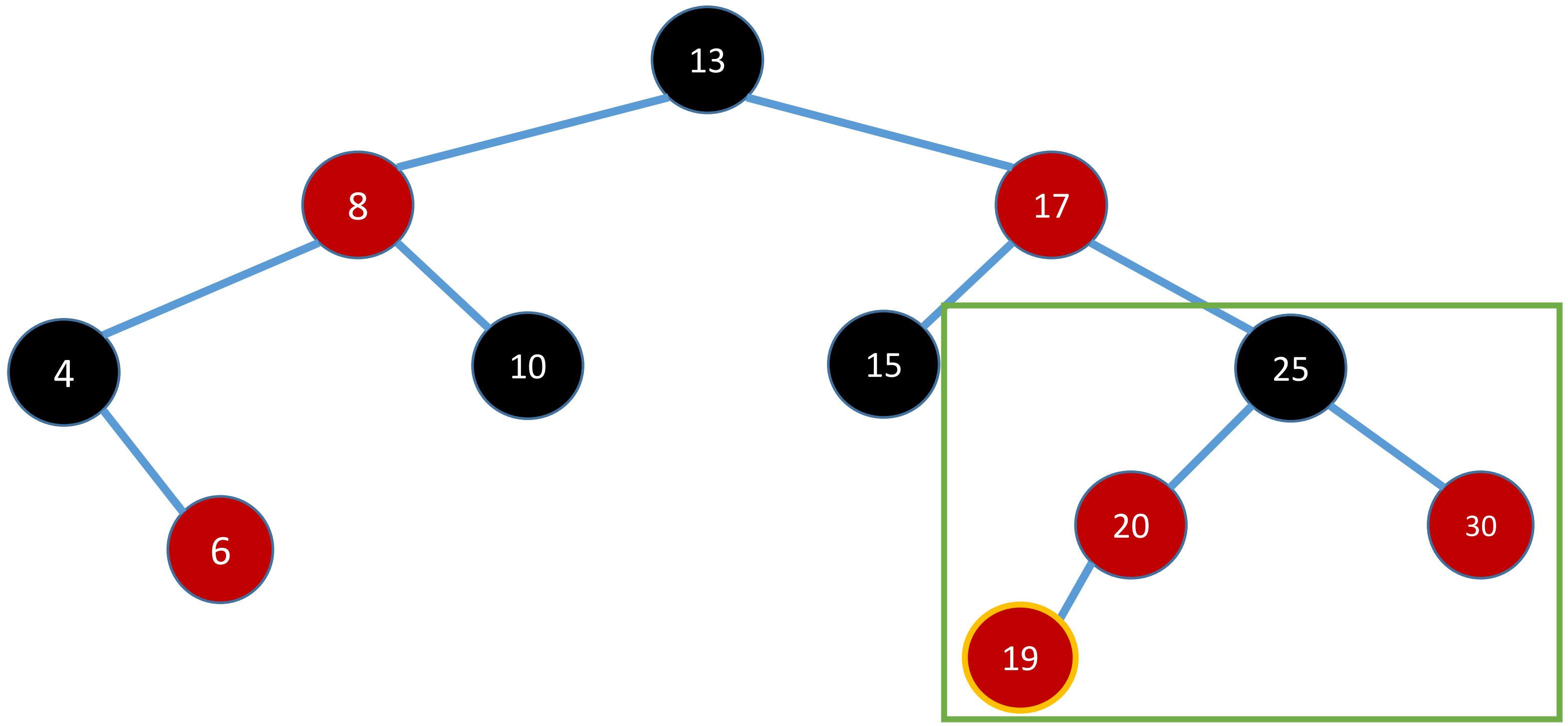
Red-Black Trees

- 19 is added as a red node



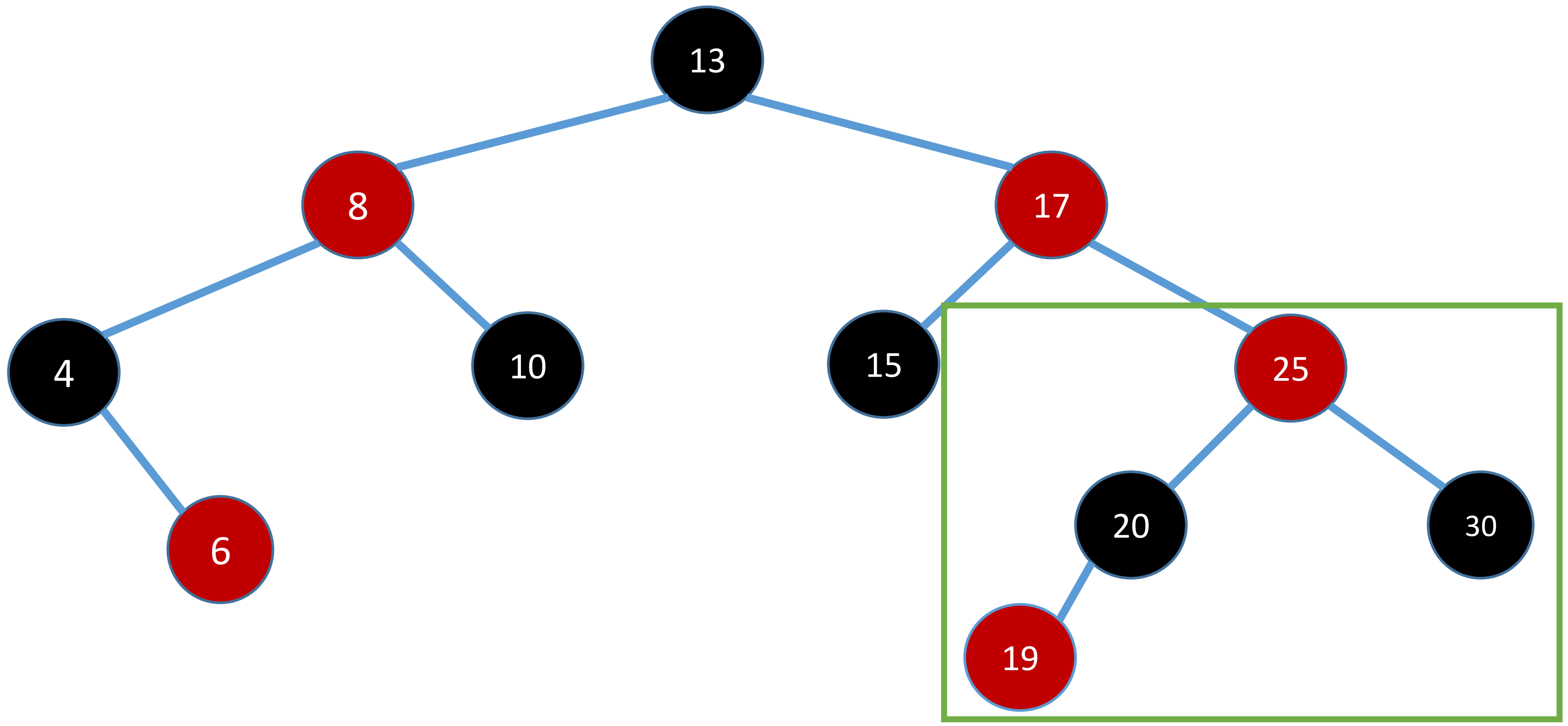
Red-Black Trees

- Scenario 2



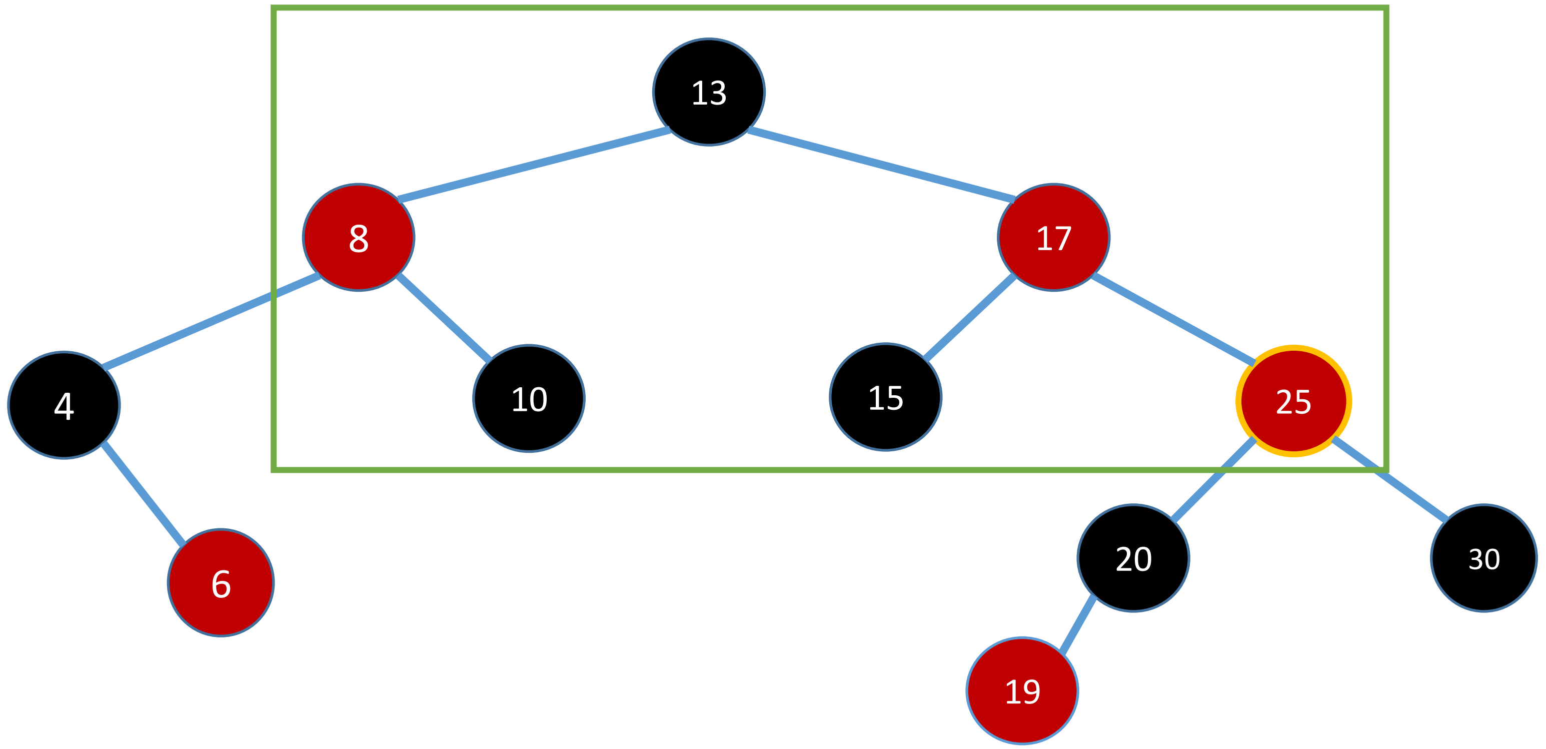
Red-Black Trees

- Recolored



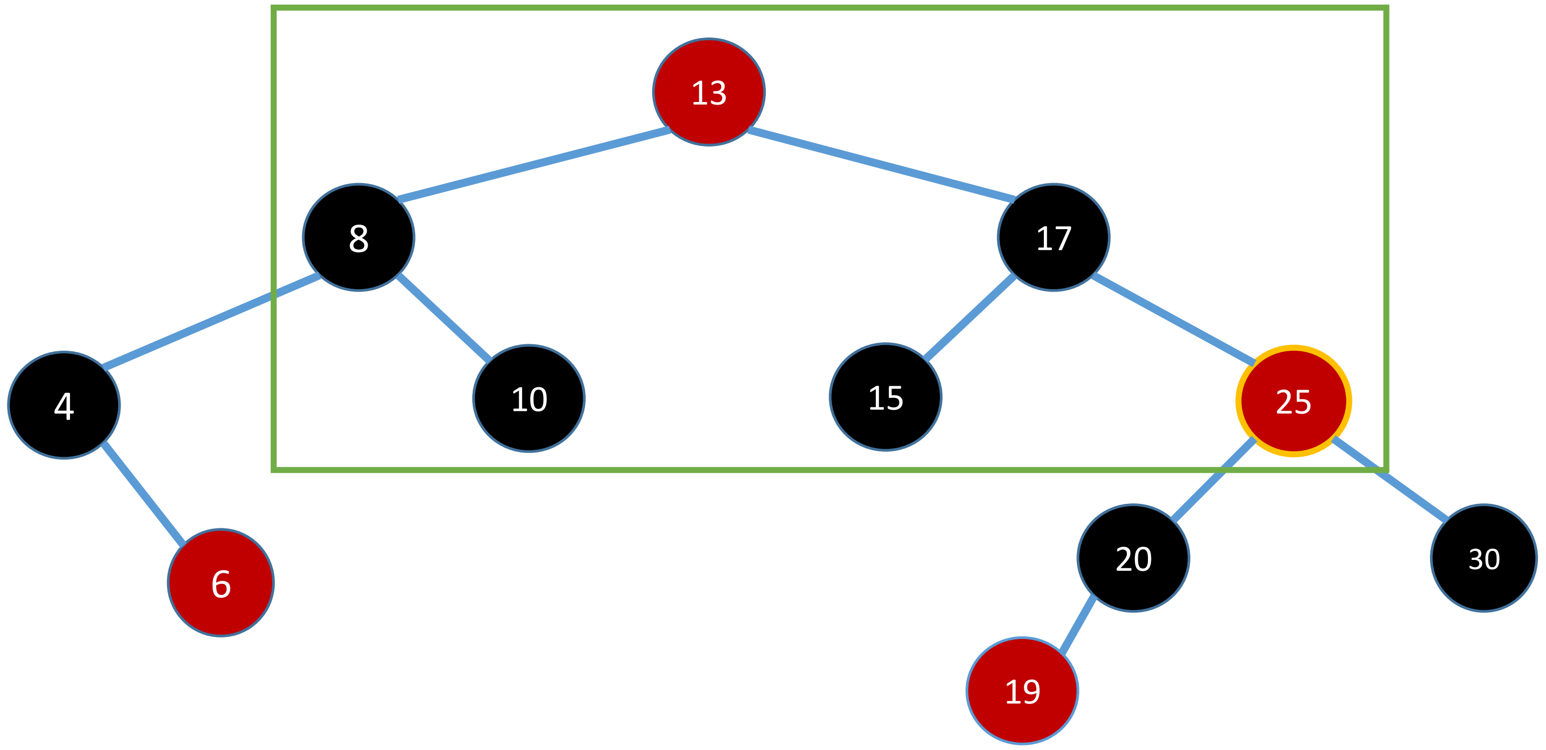
Red-Black Trees

- Scenario 2



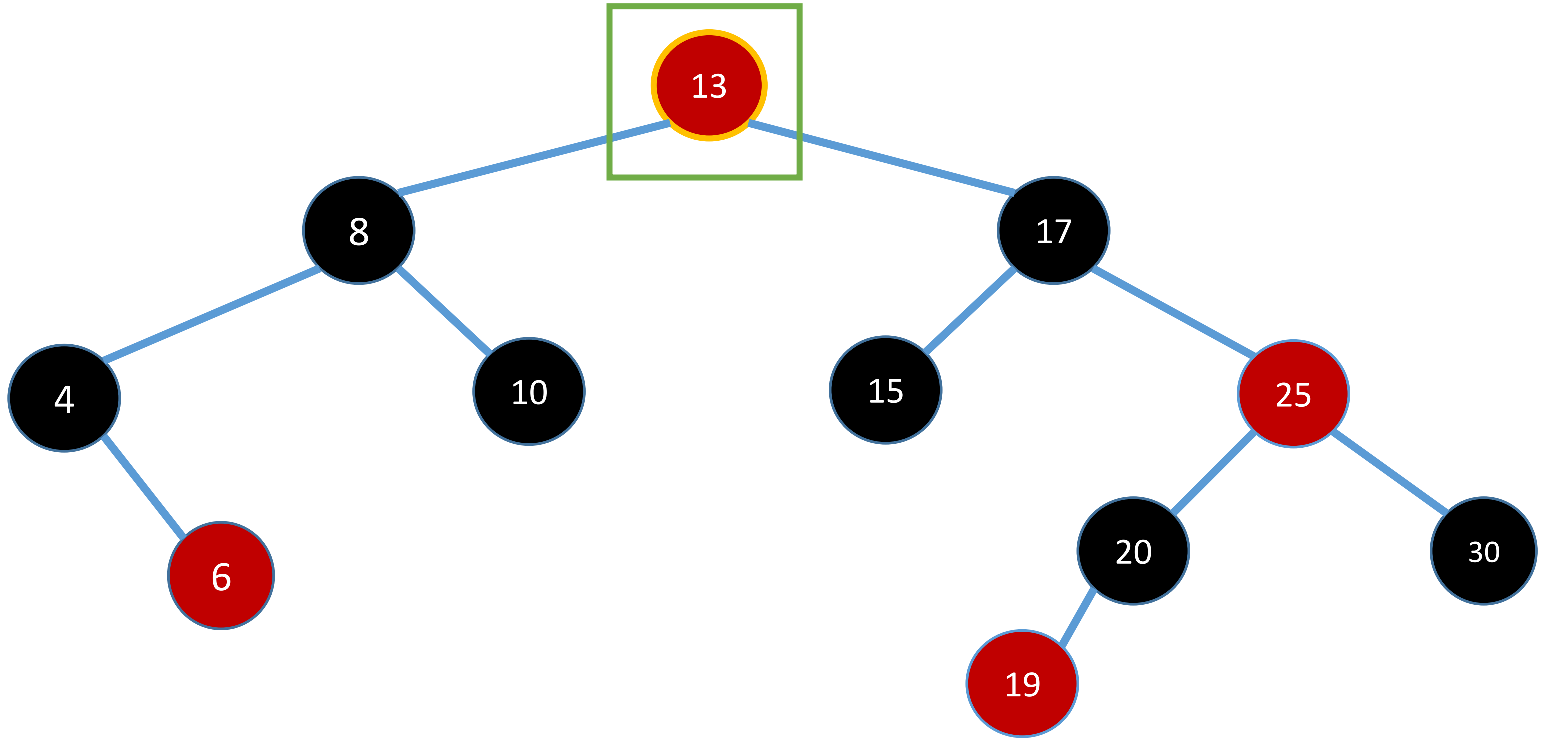
Red-Black Trees

- Recolored



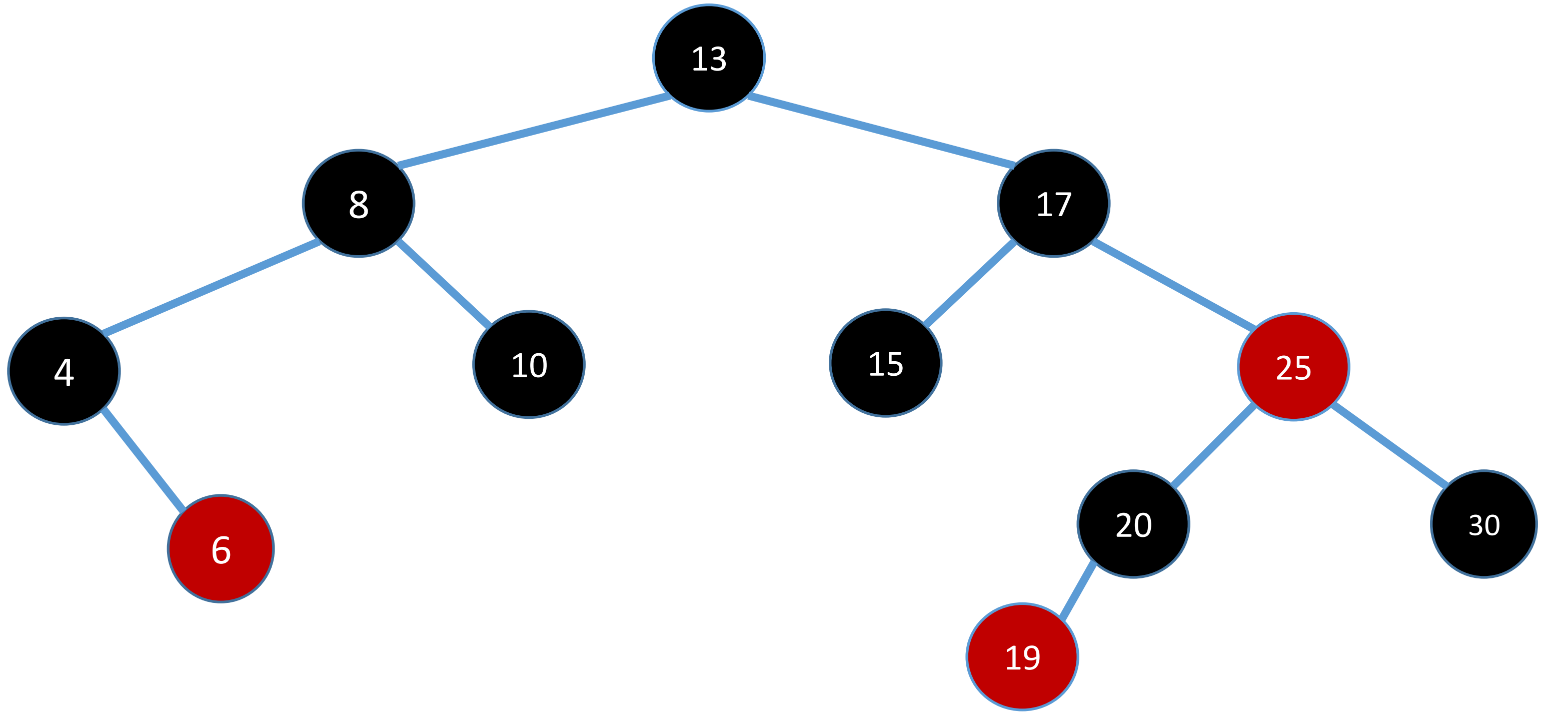
Red-Black Trees

- Root node needs recoloring



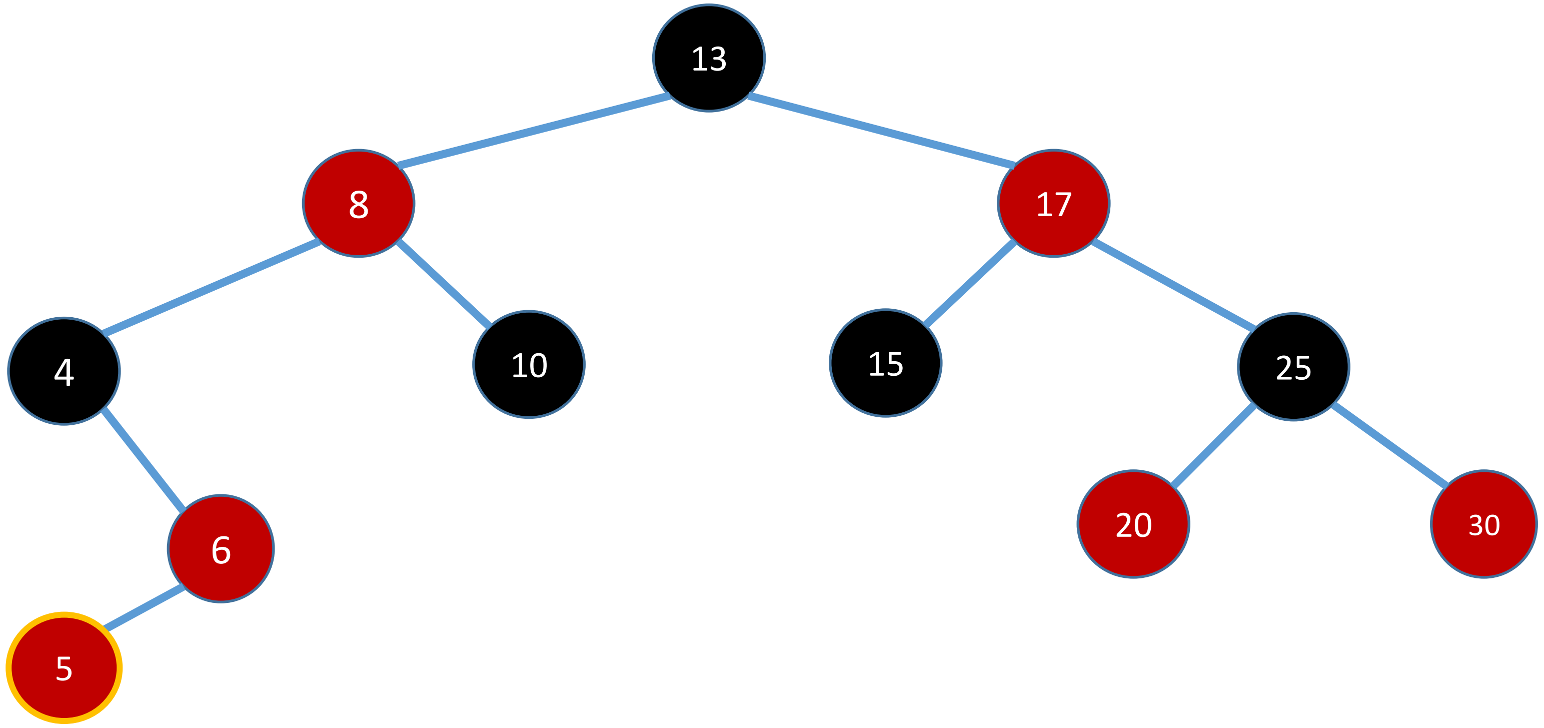
Red-Black Trees

- Finished



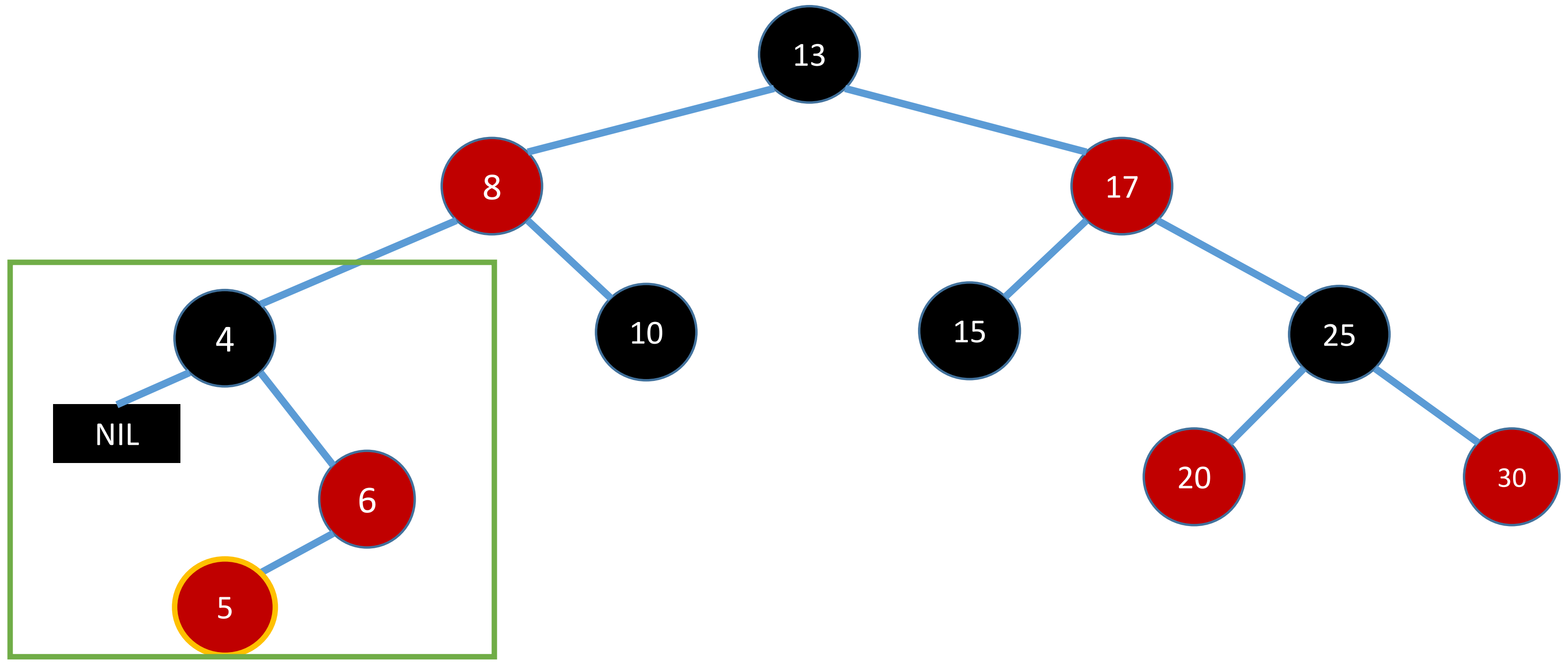
Red-Black Trees

- To demonstrate again, we'll reset to the original tree and insert 5, added as a new red node



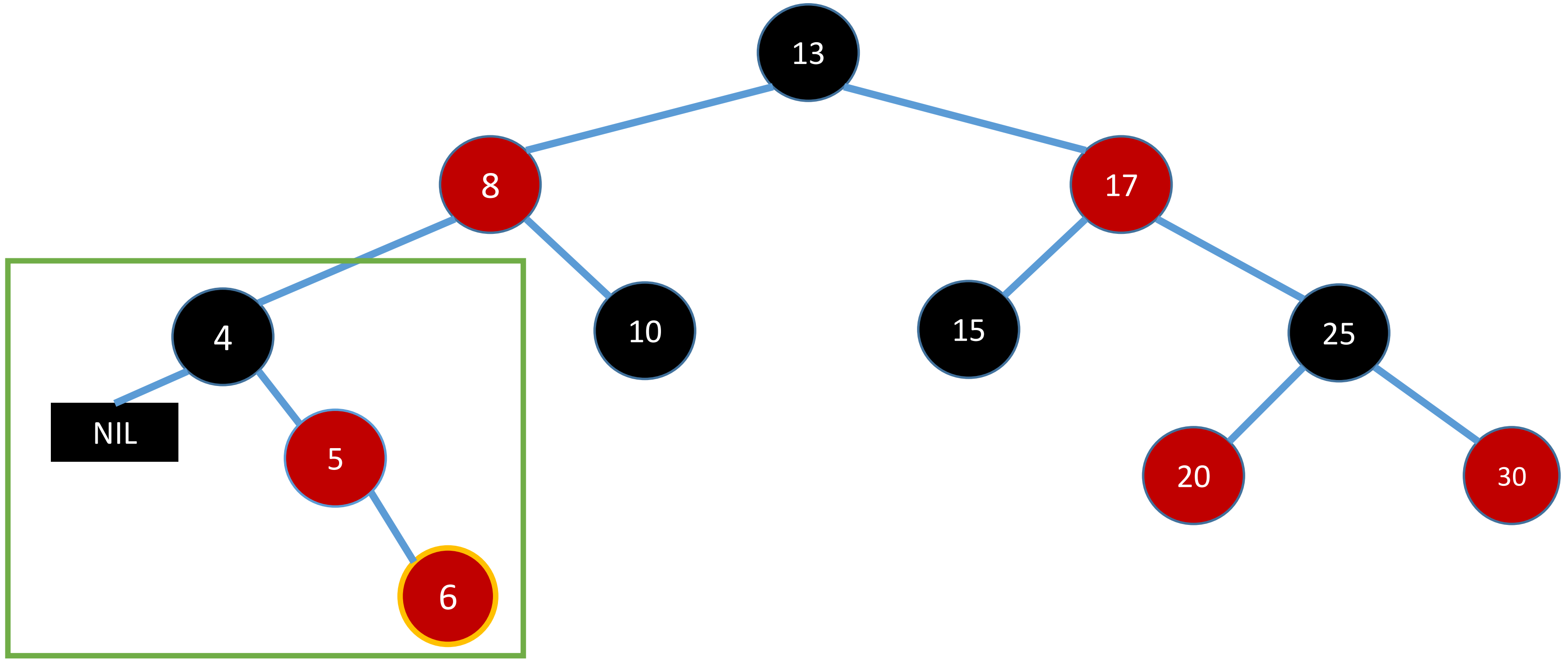
Red-Black Trees

- Scenario 3



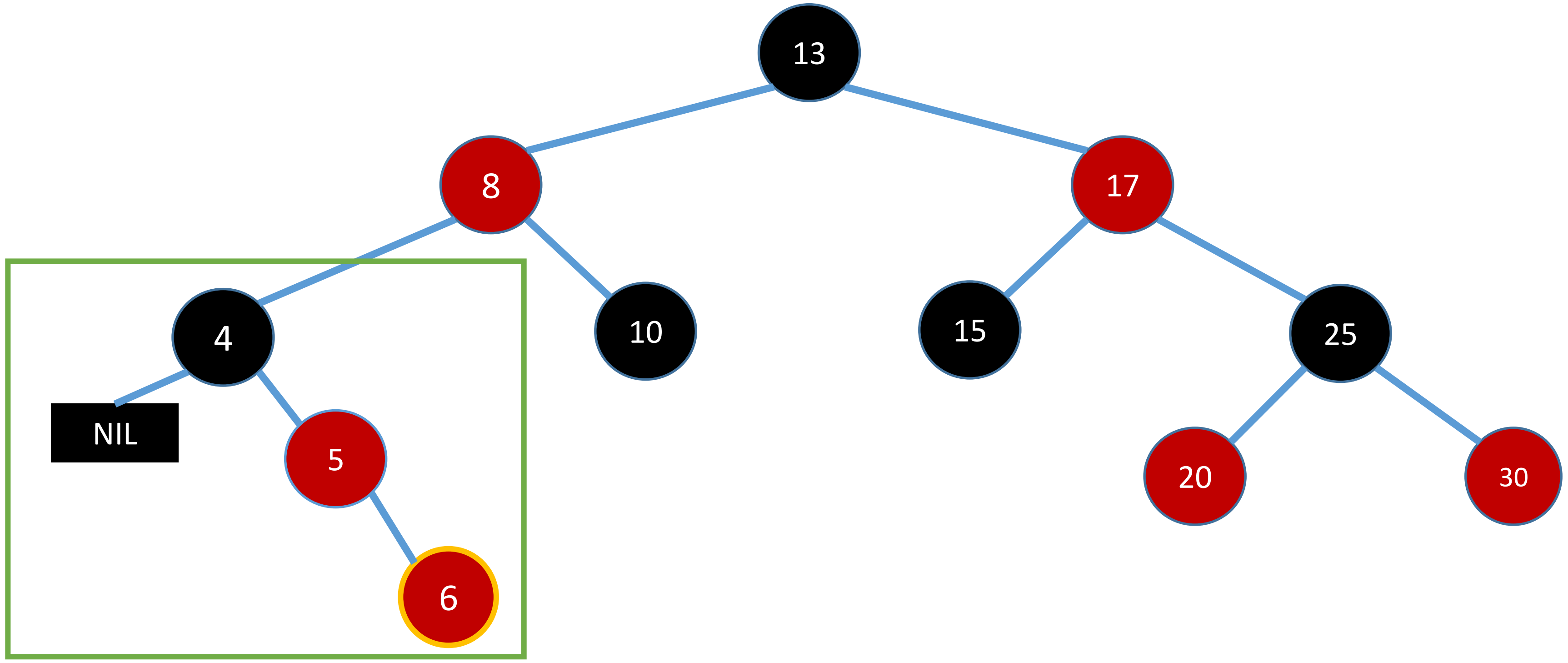
Red-Black Trees

- Node’s parent (6) is rotated



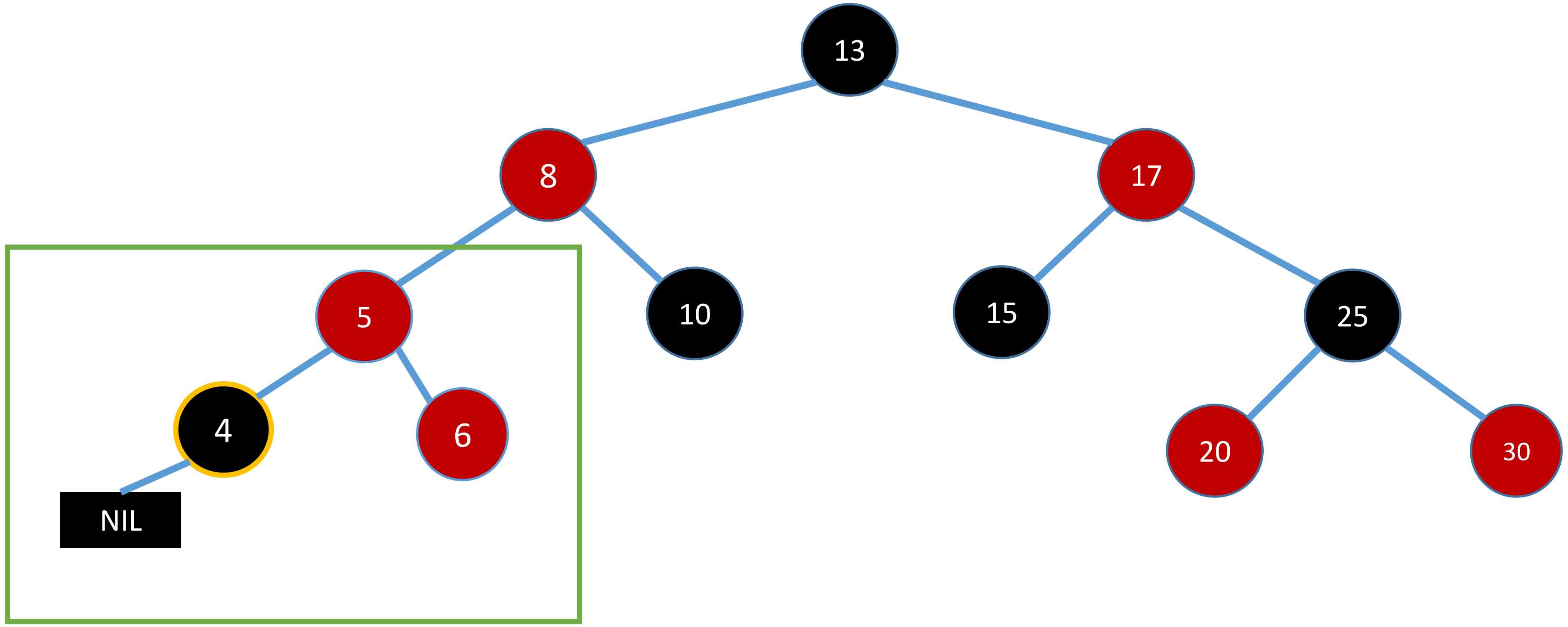
Red-Black Trees

- Scenario 4



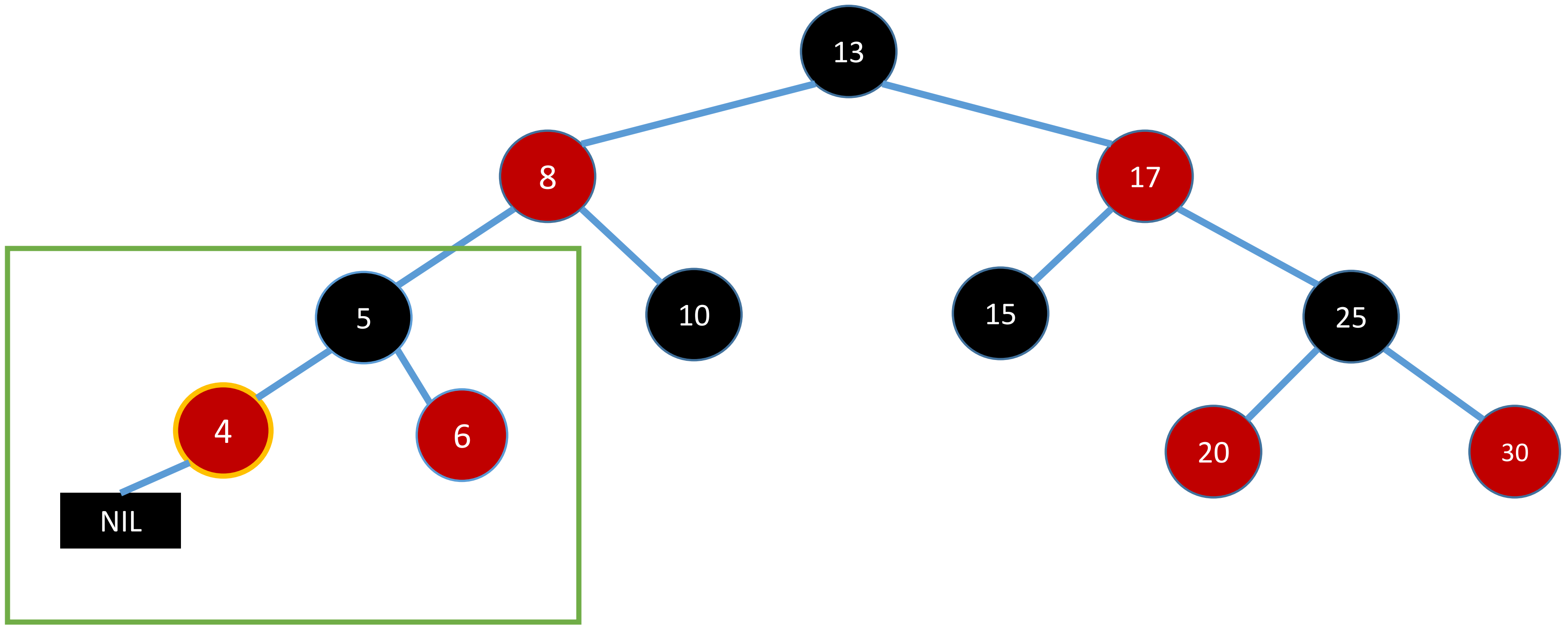
Red-Black Trees

- Node’s grandparent (4) is rotated



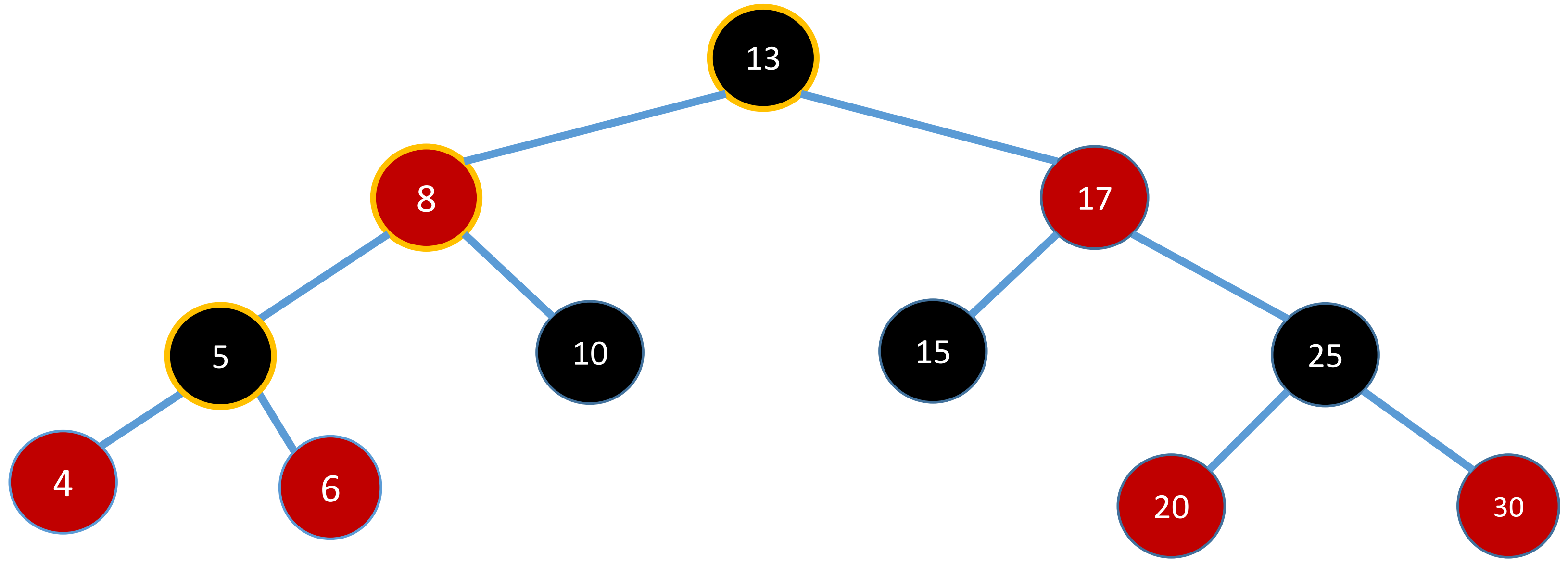
Red-Black Trees

- Original parent and grandparent are recolored



Red-Black Trees

- Moving up the tree to the root, no other violations are found
  - Finished

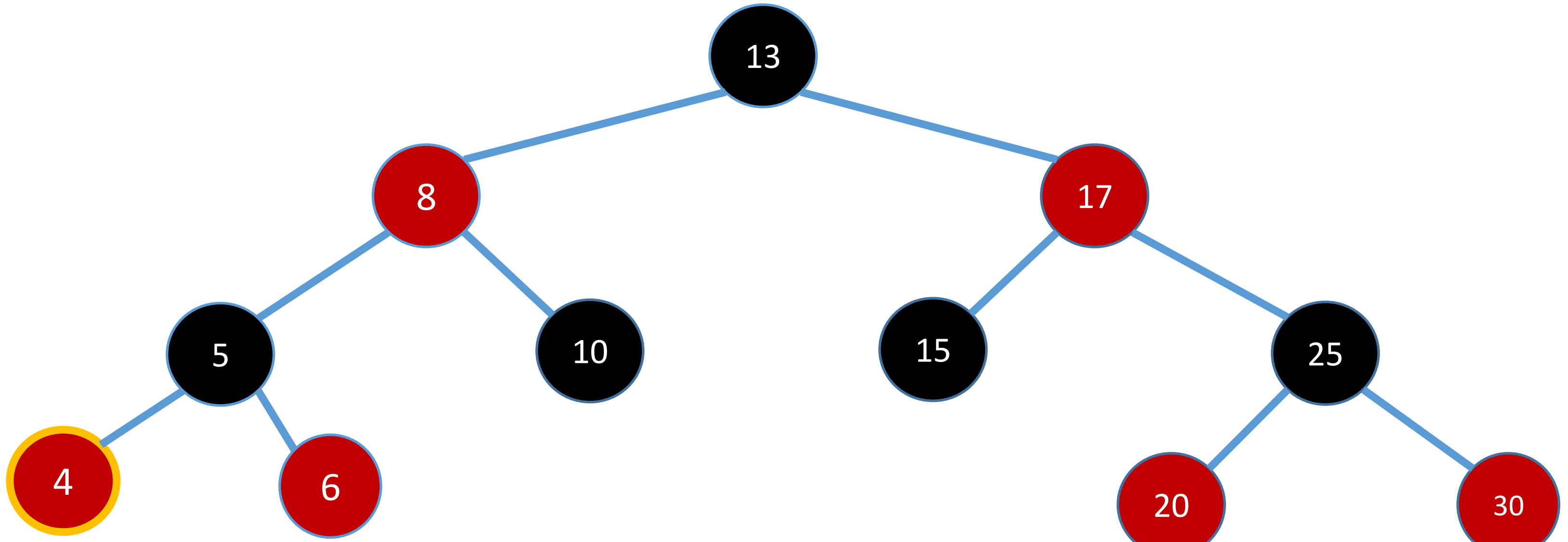


Removal

- Removing a red node is simple because:
  - The root will never be red
  - A red node won't have a red child, so no possibility of violating the no-adjacent-reds rule
  - Will not have changed the black height of a node
- Removing a black node is often more complicated because:
  - The root could have been removed
  - Removing a black node could result in violating the no adjacent reds rule
  - May change the black height of a node

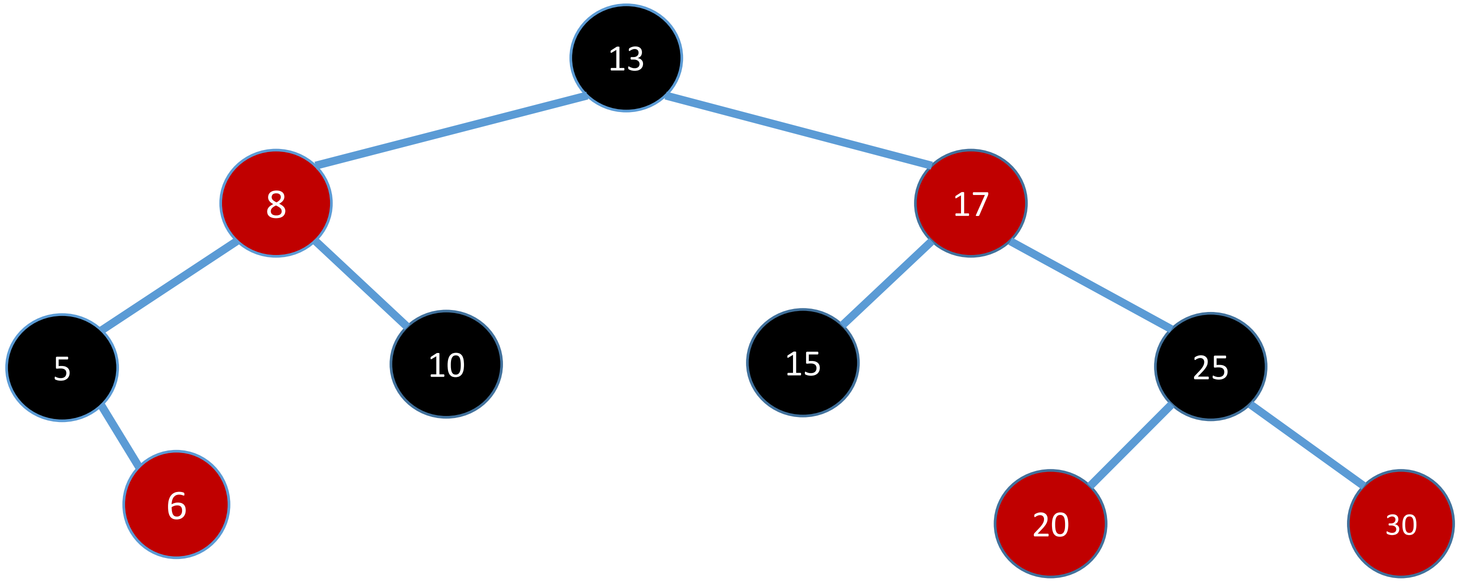
Removal

- Removing a **red leaf** (4)



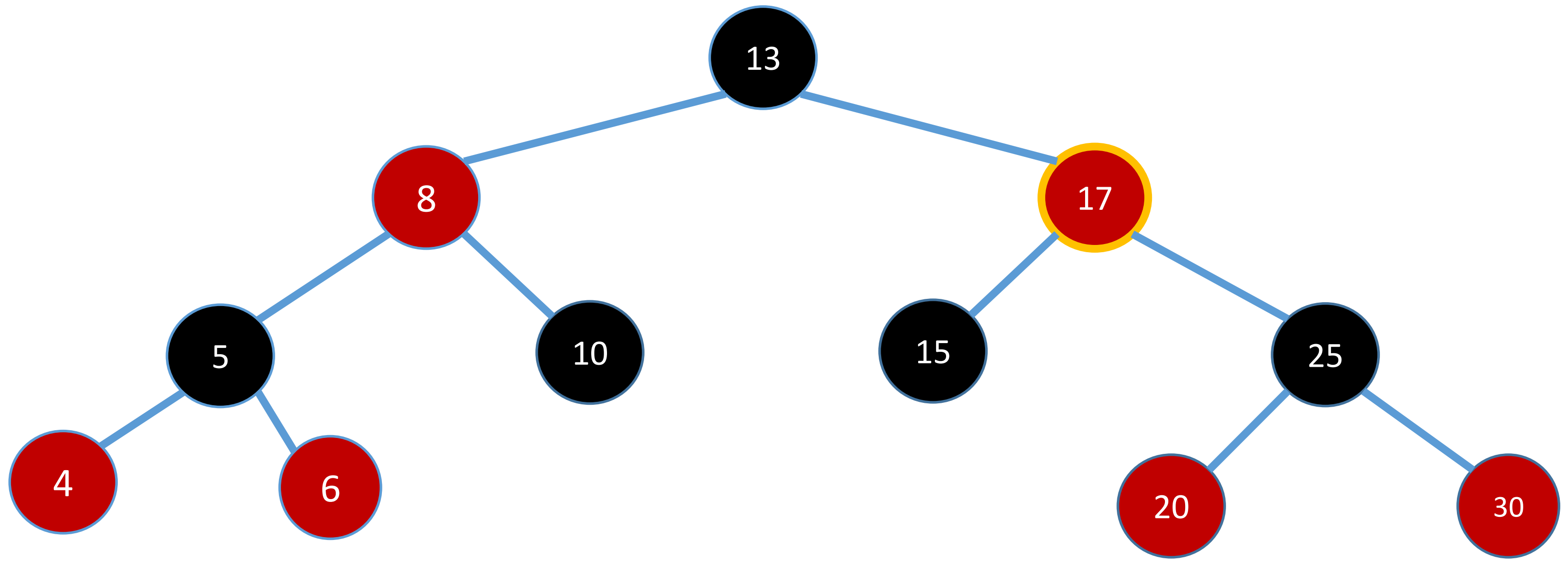
Removal

- Removing a **red leaf** (4)
  - Simply remove the node



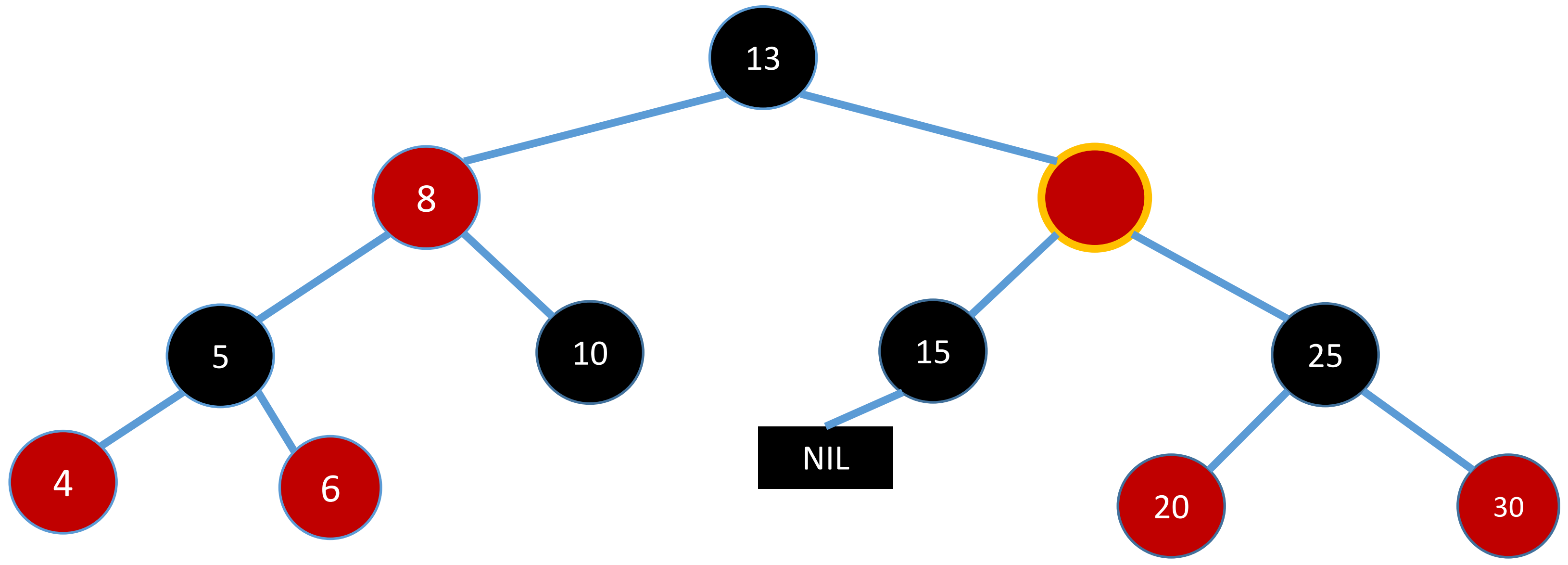
Removal

- Removing a **red node with two children** (17)
    - **(There will never be a red node with one child)**



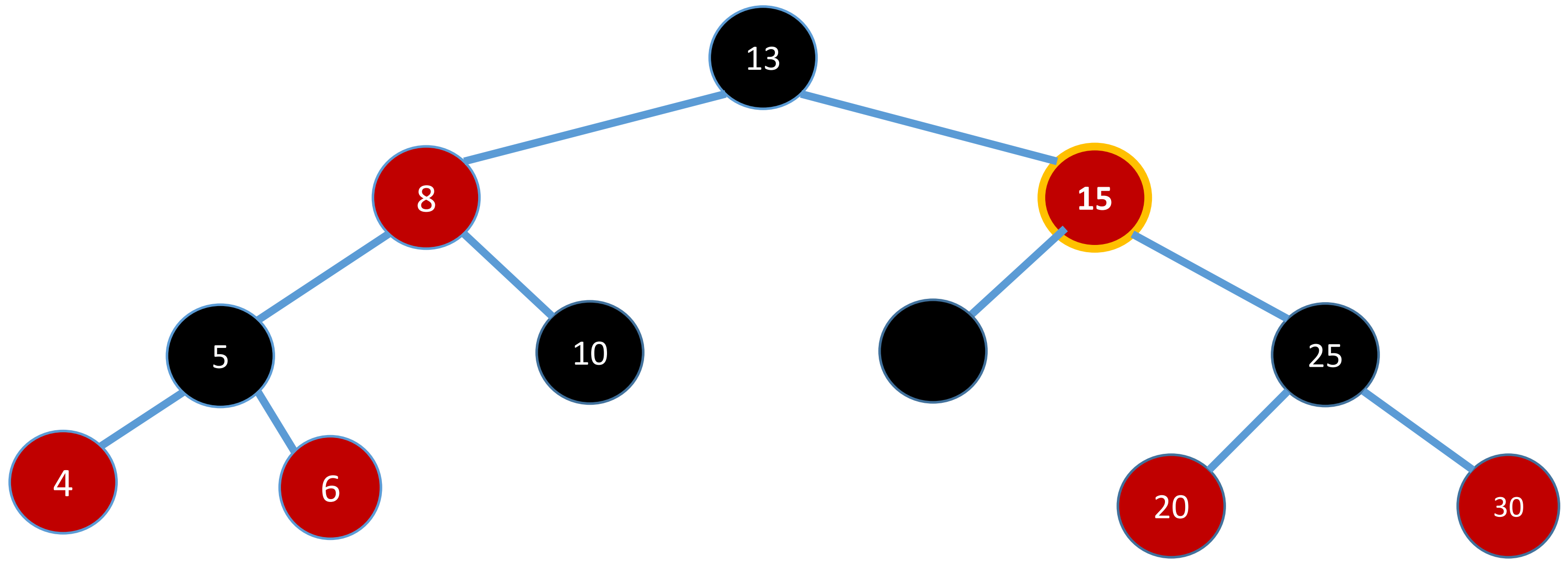
Removal

- Removing a **red node with two children** (17)
  - Value is removed



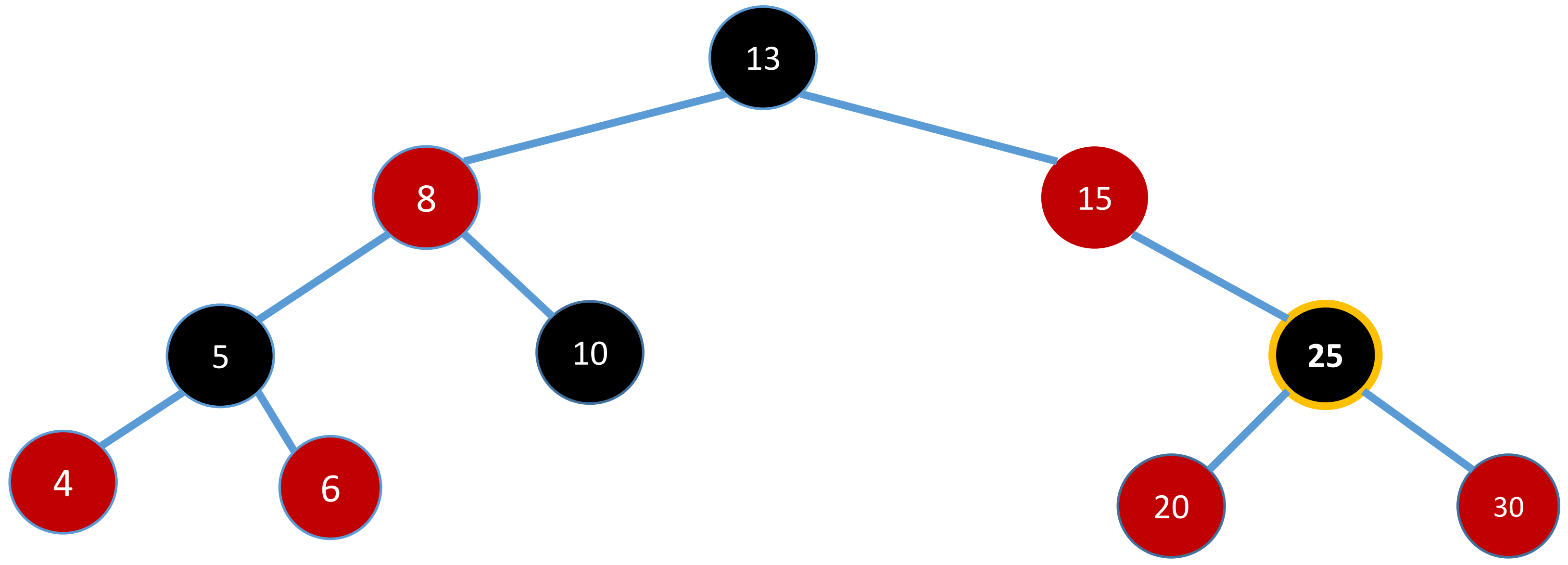
Removal

- Removing a **red node with two children** (17)
    - Promote largest value on the left side (15)

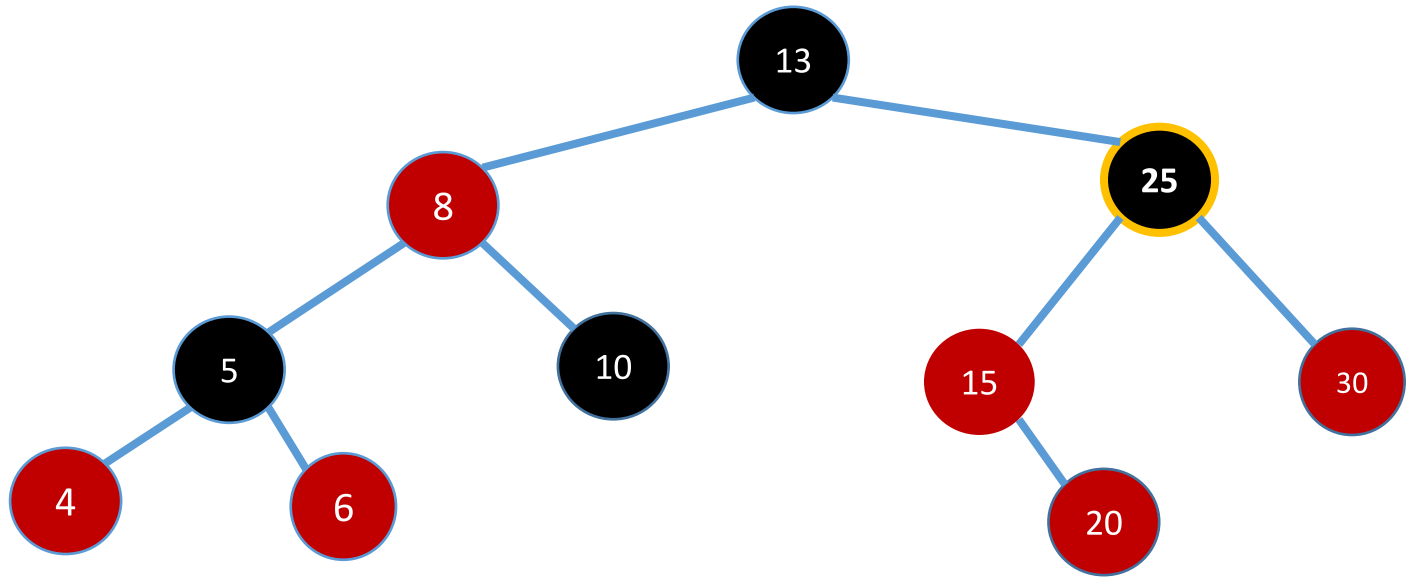


Removal

- Removing a **red node with two children** (17)
  - Rotate the removed node's sibling

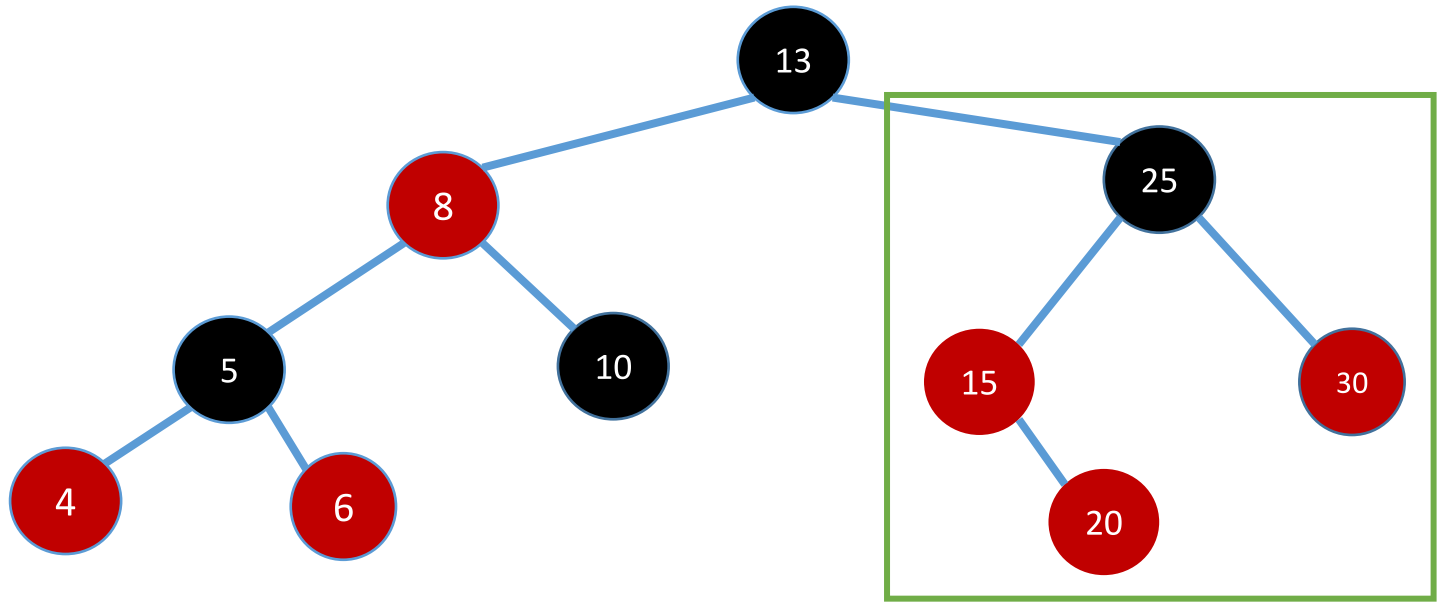


Removal

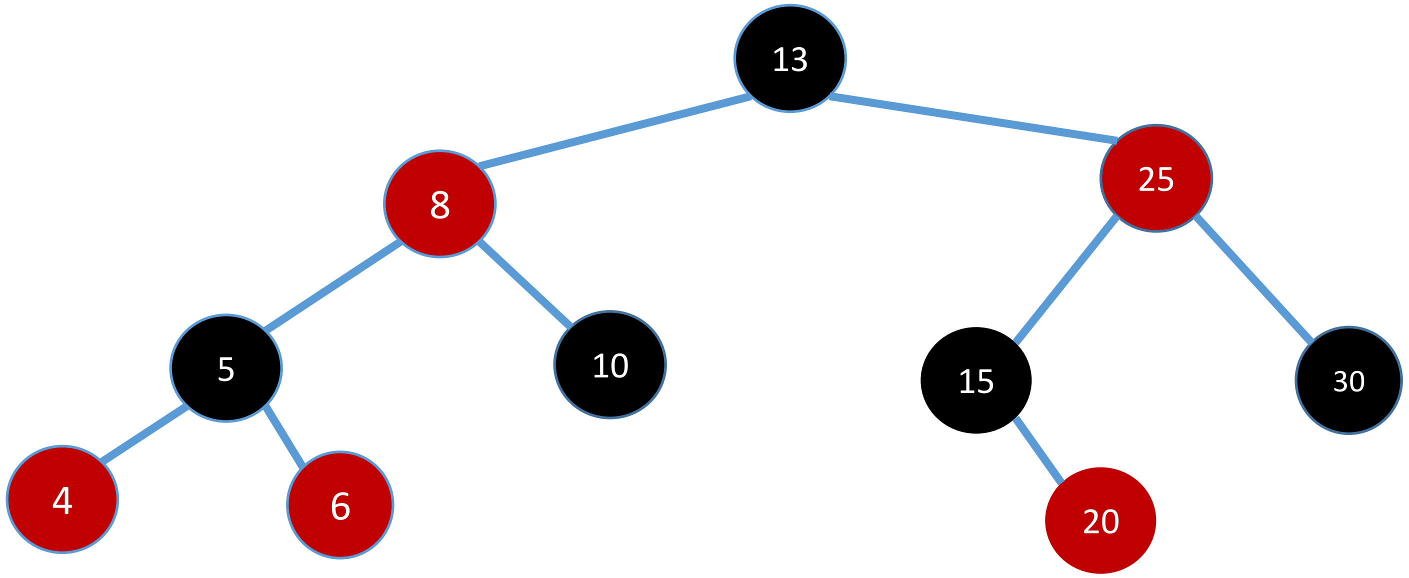


Removal

- Scenario 2
  - Recolor

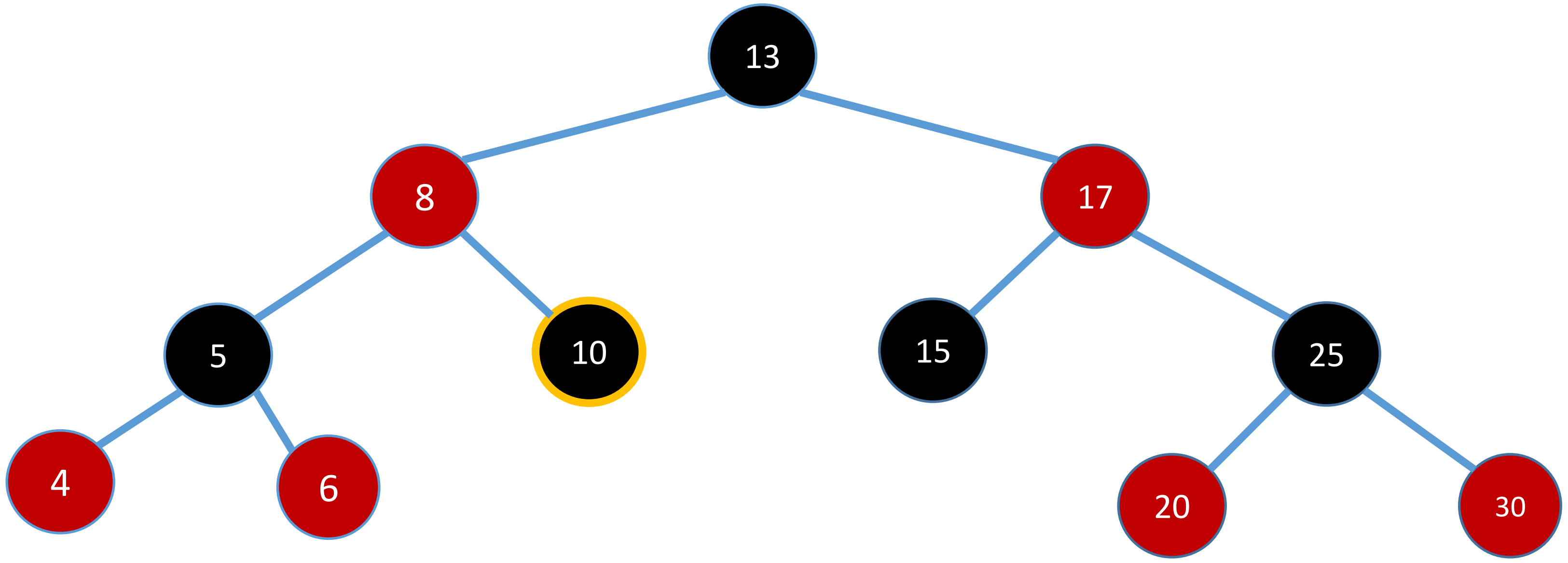


Removal



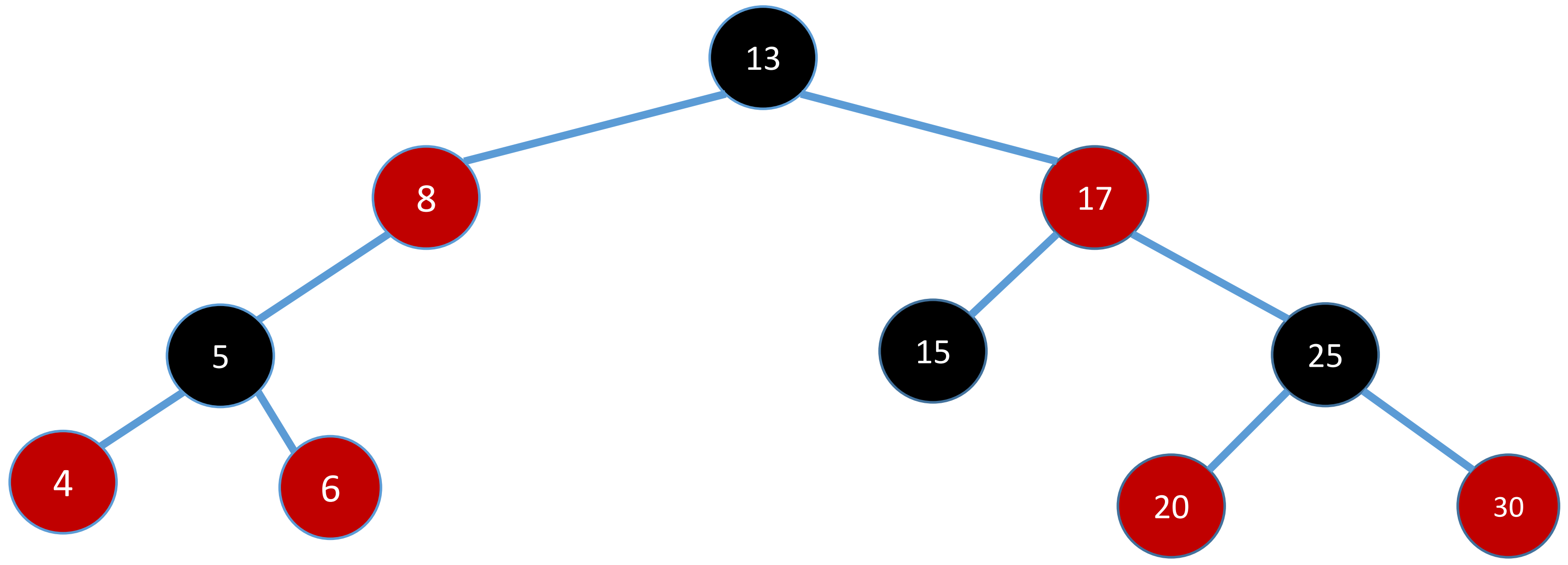
Removal

- Removing a **black leaf** (10)
  - **Its sibling is black and at least one nibling is red**



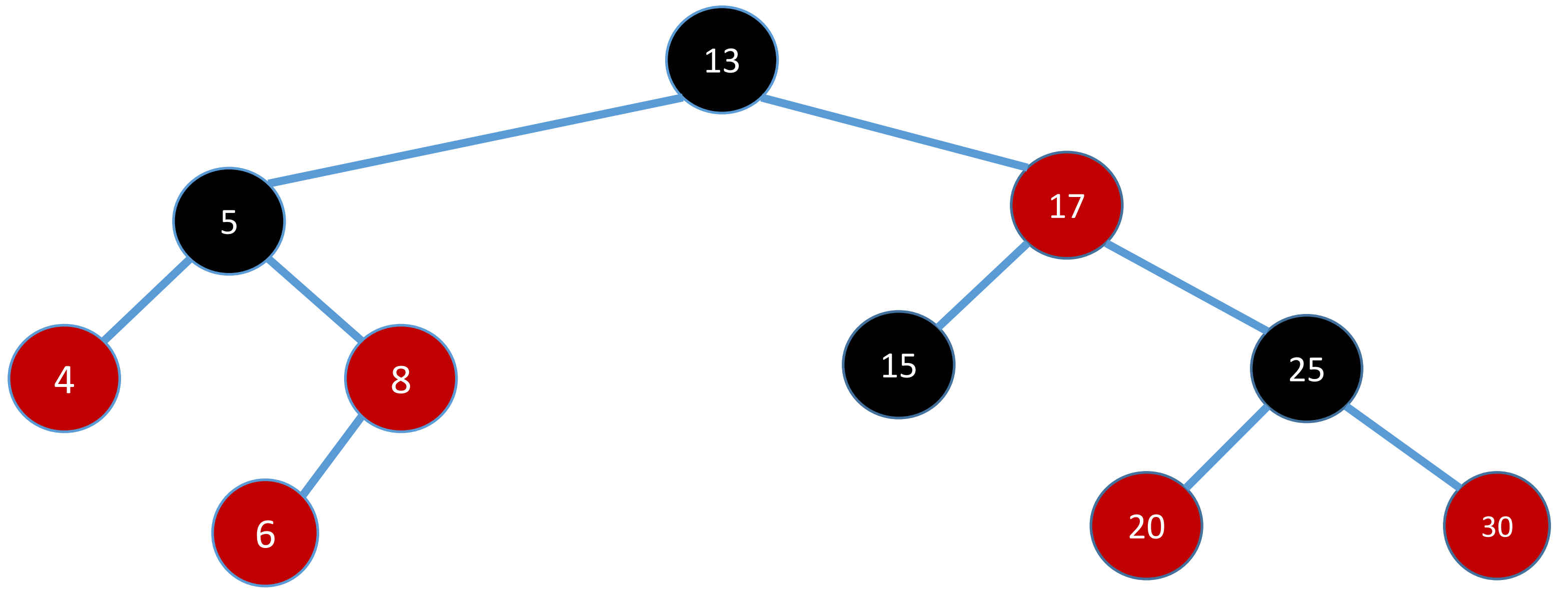
Removal

- Removing a **black leaf** (10)
  - Delete the node



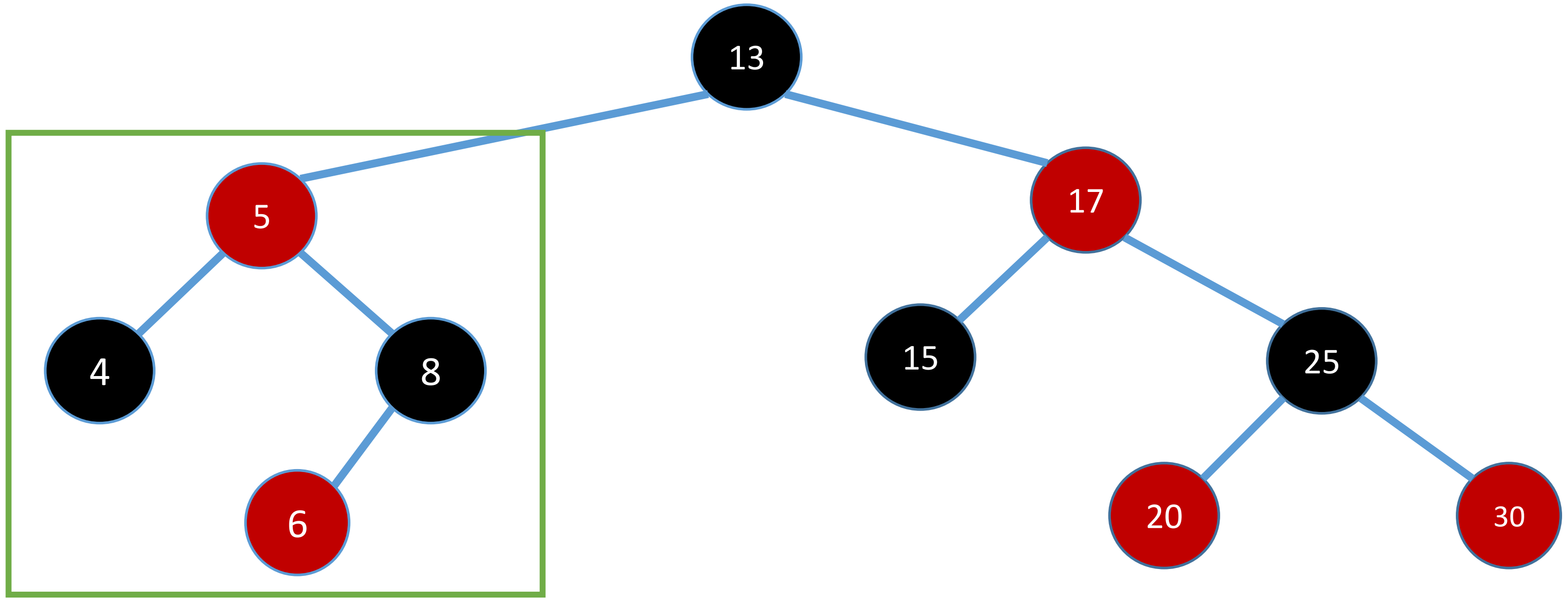
Removal

- Removing a **black leaf** (10)
  - Rotate the removed node's sibling



Removal

- Removing a **black leaf** (10)
    - Recolor (Was Scenario 2)



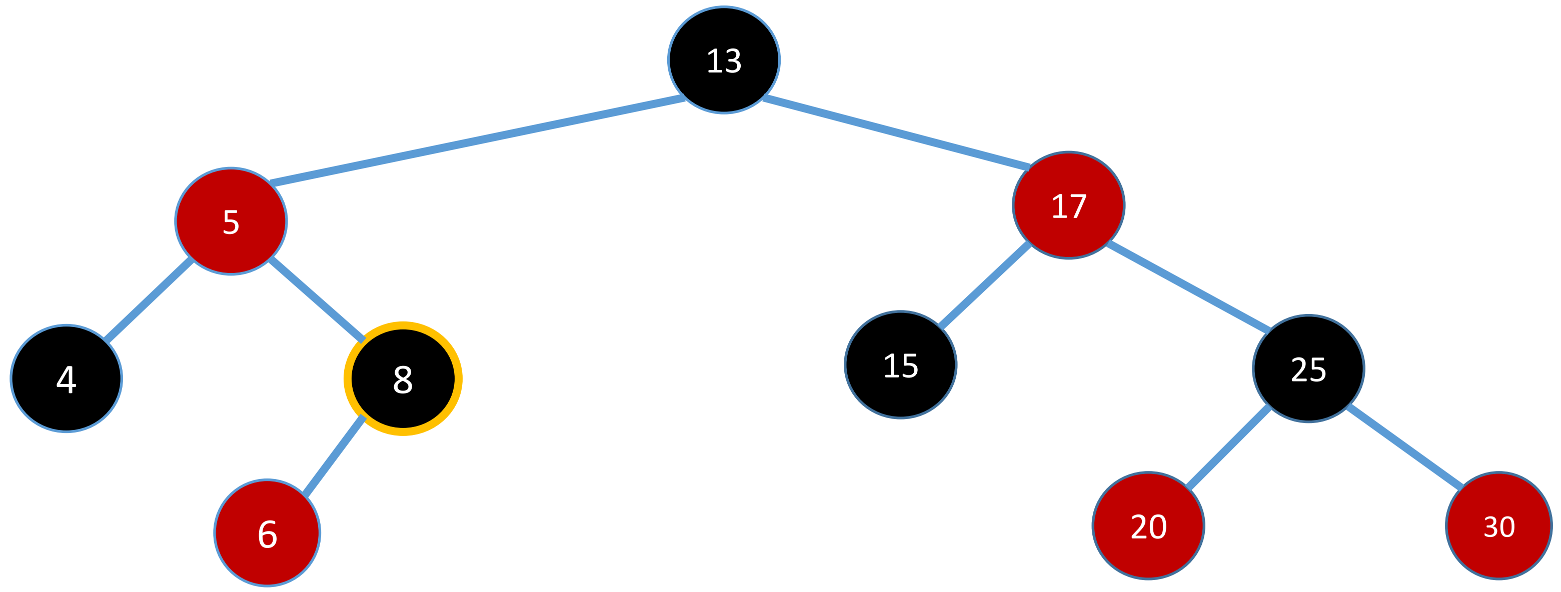
Removal

- This situation was for a black leaf that had a black sibling with a red nibling

- If the nibling(s) were black:
  - Recolor the sibling and parent
  - Any further necessary recoloring is done on its way up to the root

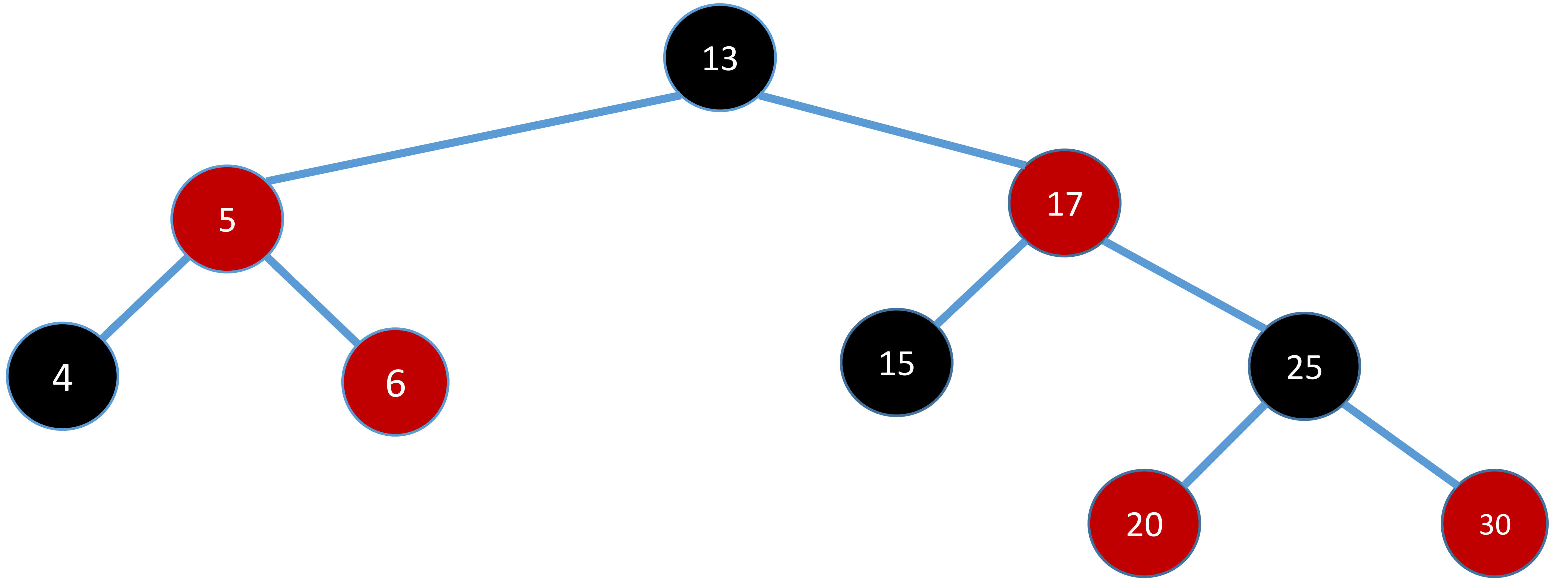
Removal

- Removing a **black node with one child** (8)



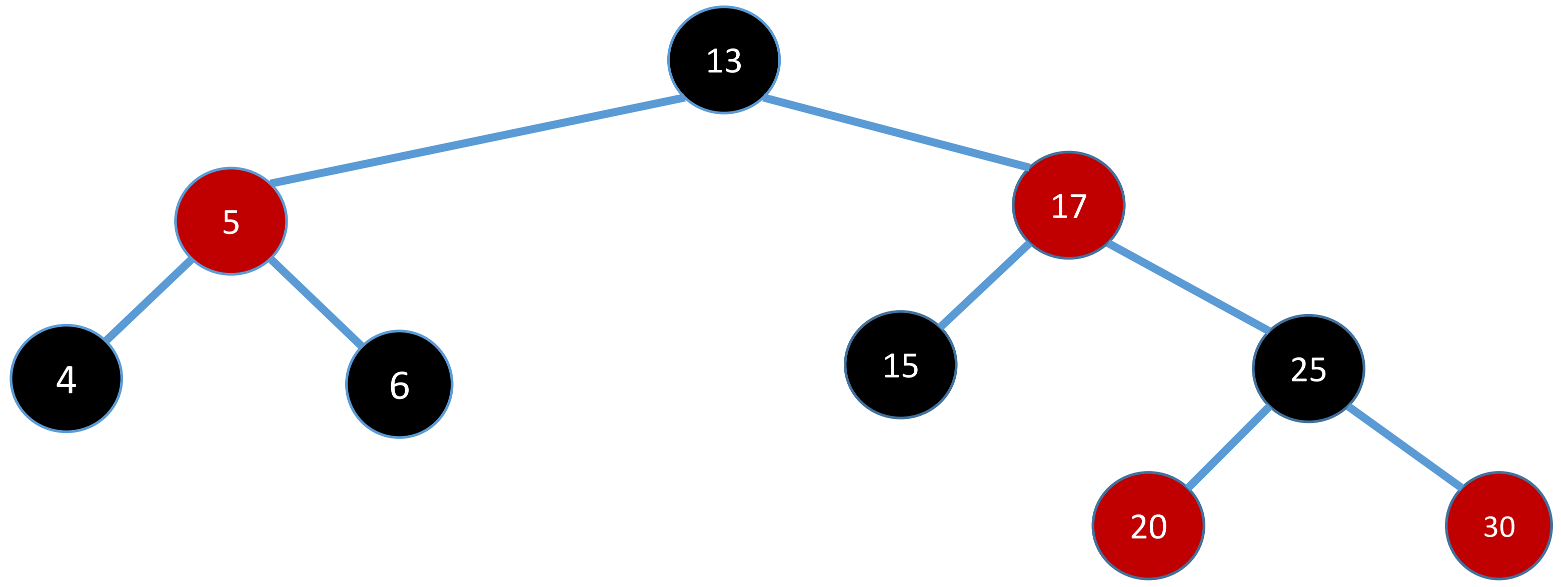
Removal

- Removing a **black node with one child** (8)
  - Replace with the child



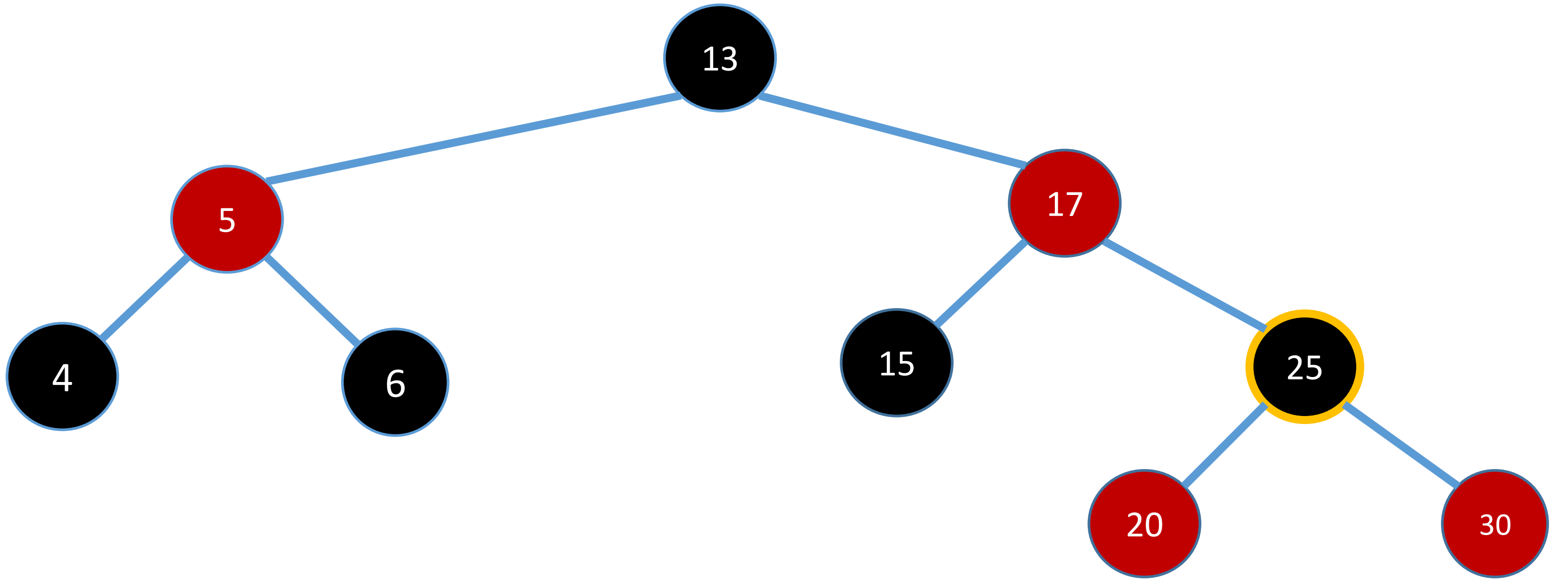
Removal

- Removing a **black node with one child** (8)
  - Recolor to black



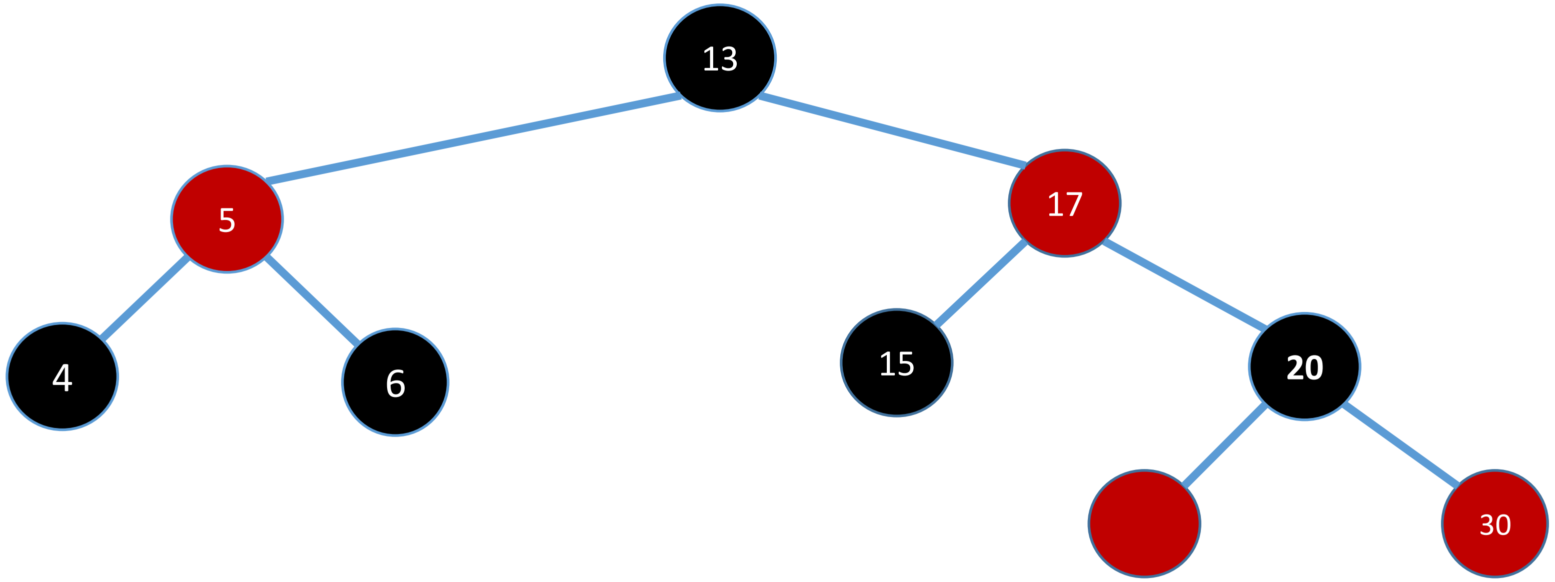
Removal

- Removing a **black node with two children** (25)



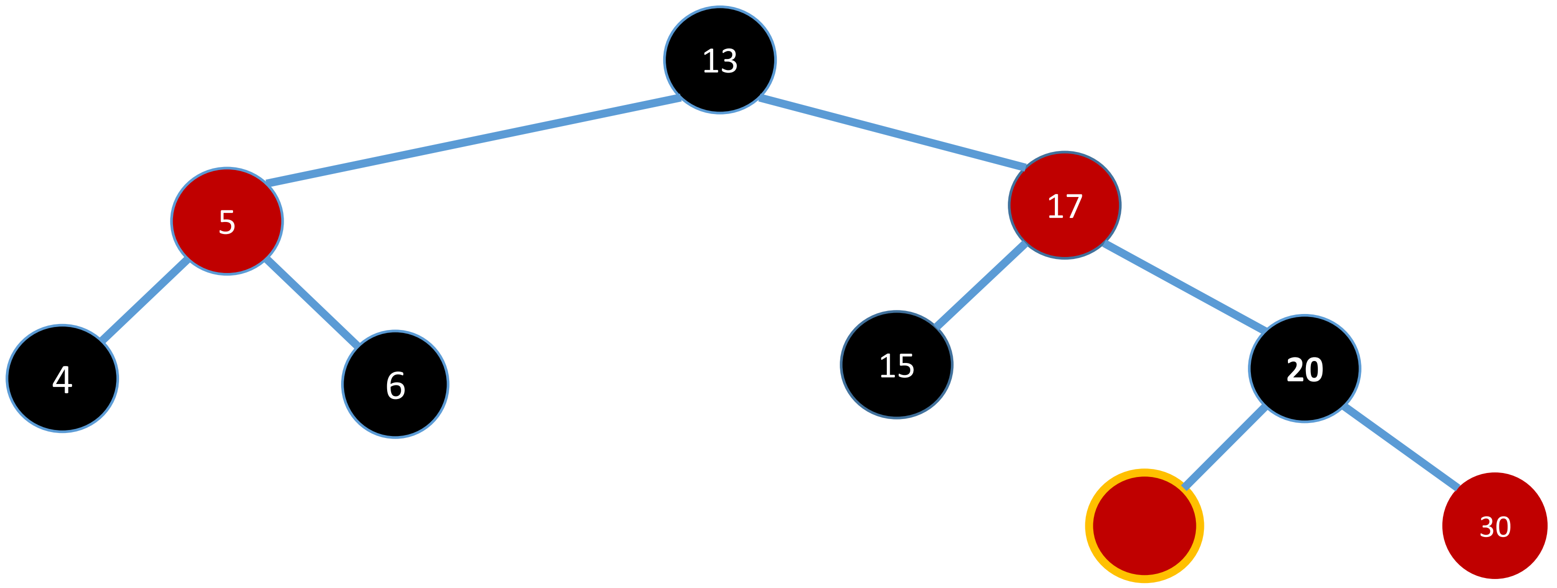
Removal

- Removing a **black node with two children** (25)
  - Promote the largest value on the left side



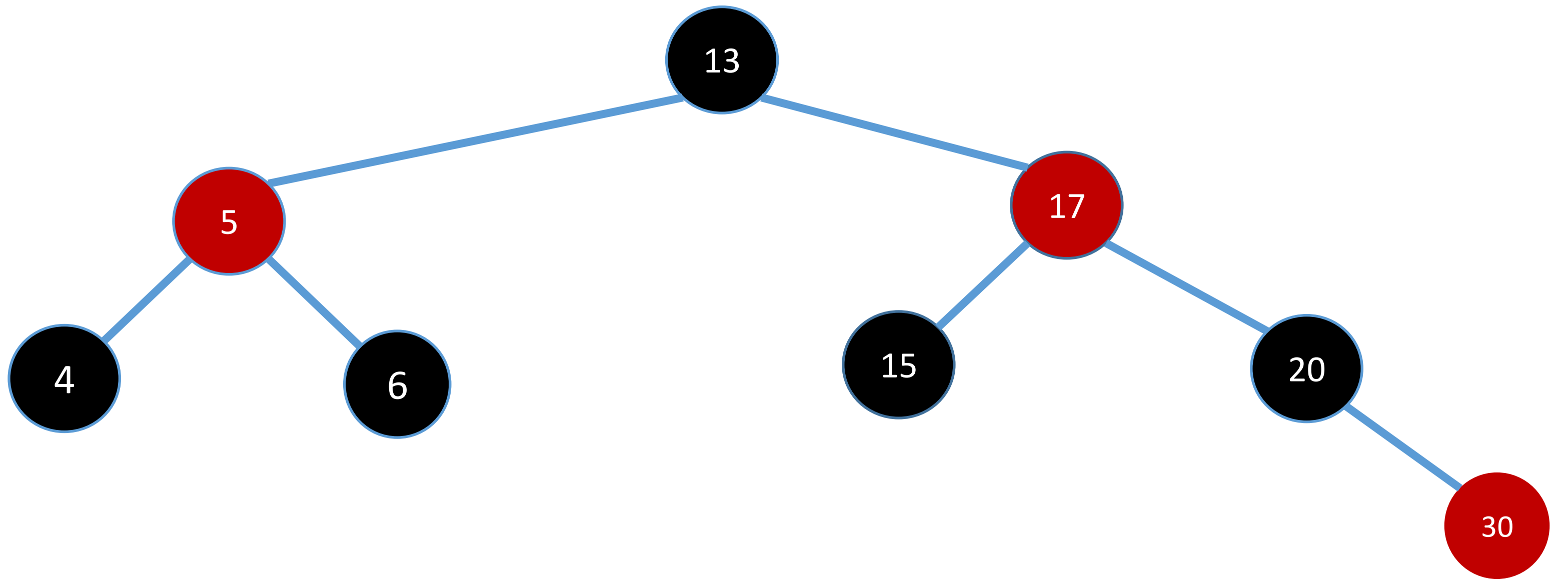
Removal

- Removing a **black node with two children** (25)
    - Was a red leaf, OK to delete



Removal

- Removing a **black node with two children** (25)
  - Finished



Complexities

- Since AVL and Red-Black trees are self-balancing, their complexity for search, insertion, and removal are O(log n)

- Rotation (and recoloring for Red-Black Trees) are constant time operations: O(1)

- Compared to an ordinary BST, this adds some additional processing for insertion and removal.
  - However, ordinary BSTs do not guarantee O(log n) searching/insertion/removal