Red-Black Trees
(A Balanced BST)

https://cs.pomona.edu/classes/cs140/

Some notes taken from

http://www.geeksforgeeks.org/

Outline

Topics and Learning Objectives

- Discuss tree balancing (rotations, insertions, deletions)
- Prove the balancing characteristic of red-black trees
- Discuss the running time of red-black tree operations

Exercise

- Red-black tree activity

Extra Resources

- Introduction to Algorithms, 3rd, chapter 13

- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Implementations

Although Red-Black trees are not the most modern choice, they do appear in

- Java: TreeMap<K,V>
- C++: std::map

Balanced Binary Search Trees

- Why is balancing important?
- What is the worst case for a binary tree?

- Balanced tree: the height of a balanced tree stays O(lg n) after insertions and deletions

- Many different types of balanced search trees:
  - AVL Tree, Splay Tree, B Tree, Red-Black Tree

Red-Black Trees Invariants

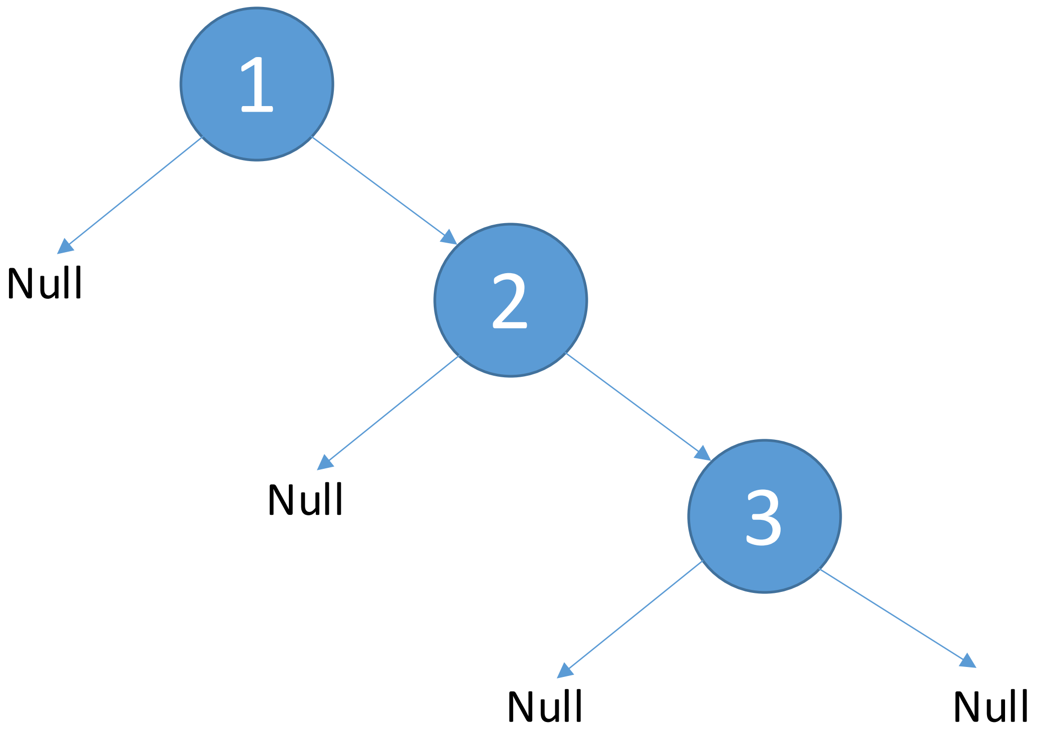
1. Each node must be labeled either red or black
2. The root must be labeled black
3. The tree cannot have two red nodes in a row (for any red node its parent, left, and right must be black)
4. Every root-NULL path must include the same number of black nodes

Can a Red-Black tree of any height have only black nodes?

Root-NULL Paths

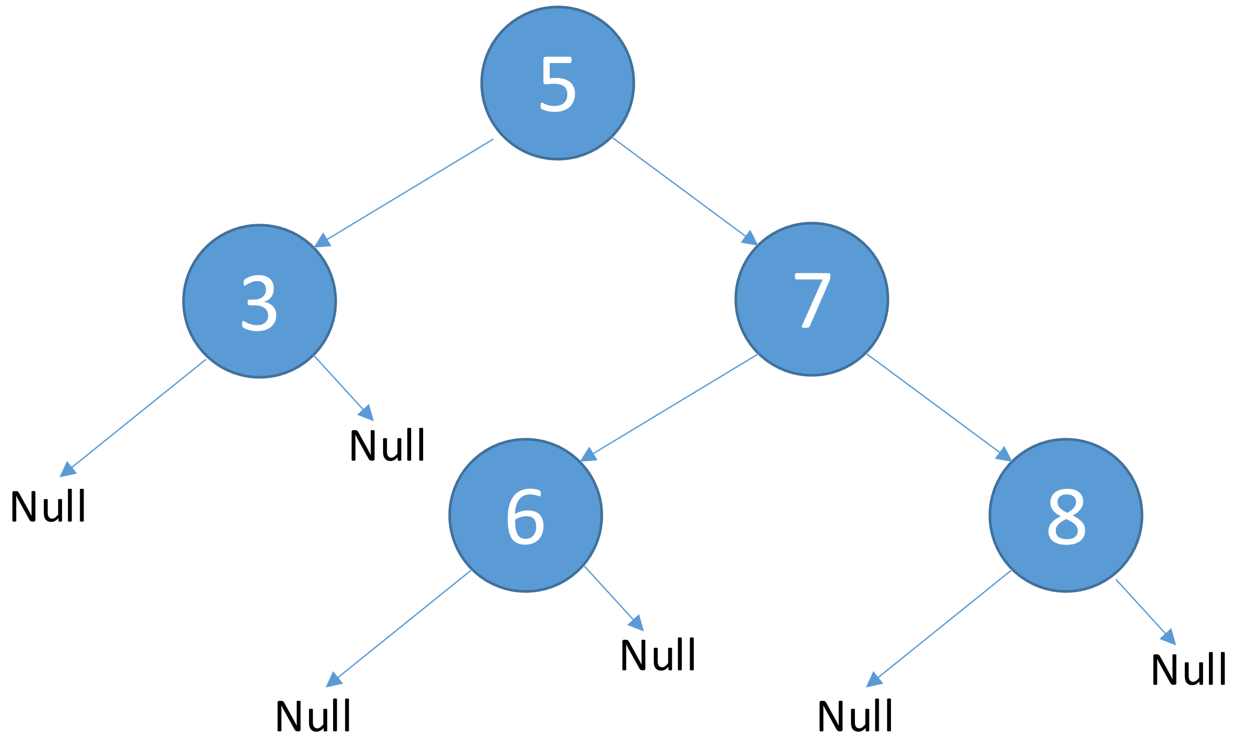
Red-Black Trees

Can a "chain" be a red-black tree?



1. Each node must be labeled either red or black
2. The root must be labeled black
3. The tree cannot have two red nodes in a row (for any red node its parent, left, and right must be black)
4. Every root-NULL path must include the same number of black nodes

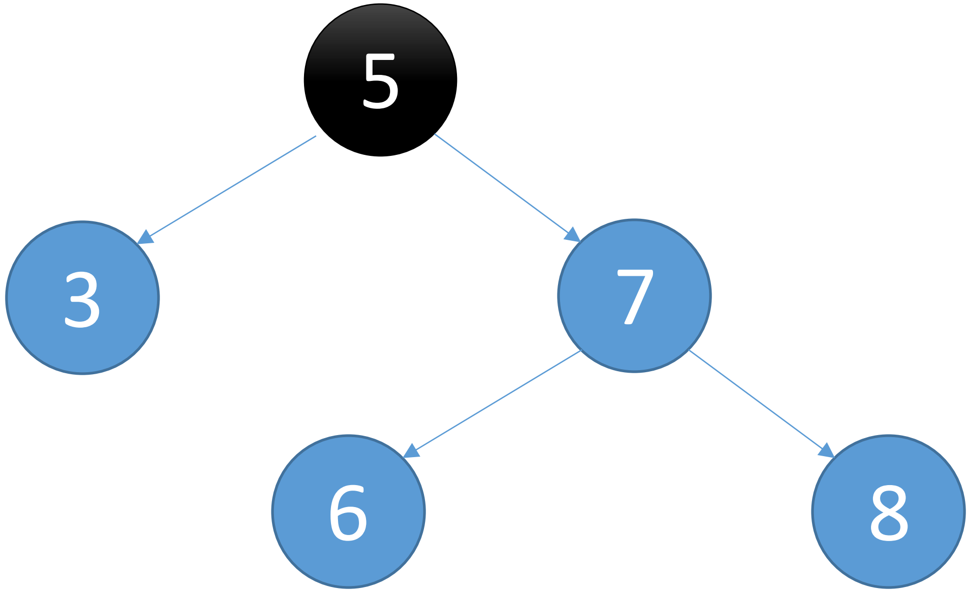
Red-Black Trees



1. Each node must be labeled either red or black
2. The root must be labeled black
3. The tree cannot have two red nodes in a row (for any red node its parent, left, and right must be black)
4. Every root-NULL path must include the same number of black nodes

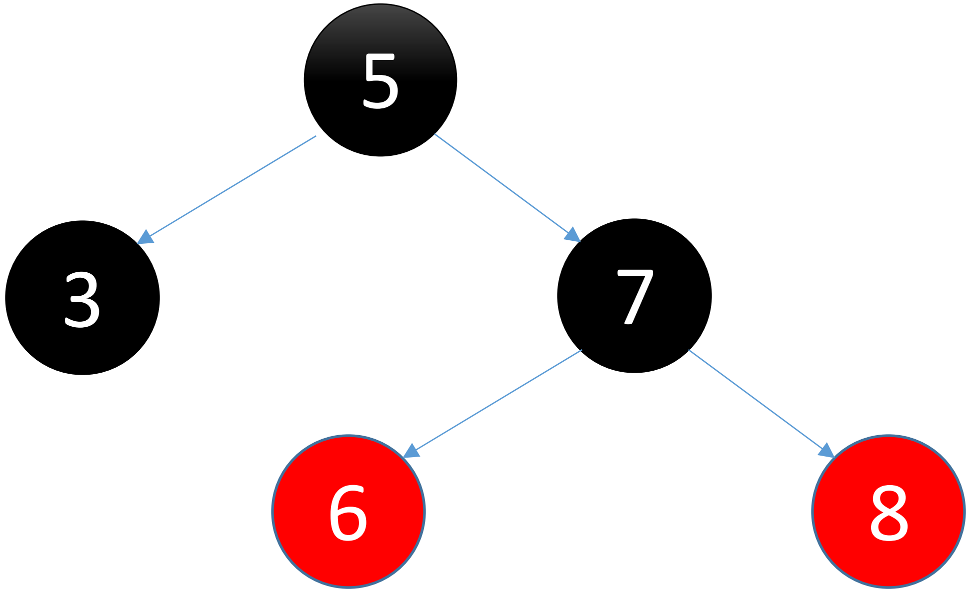
Color this as a Red-Black Tree

Red-Black Trees



1. Each node must be labeled either red or black
2. The root must be labeled black
3. The tree cannot have two red nodes in a row (for any red node its parent, left, and right must be black)
4. Every root-NULL path must include the same number of black nodes

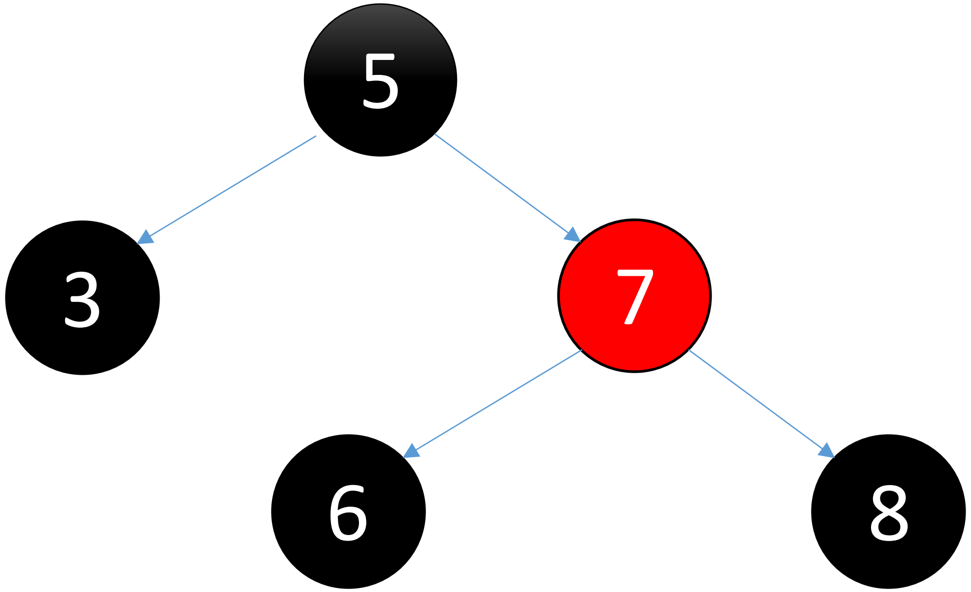
Red-Black Trees



We could also move the black color down one level

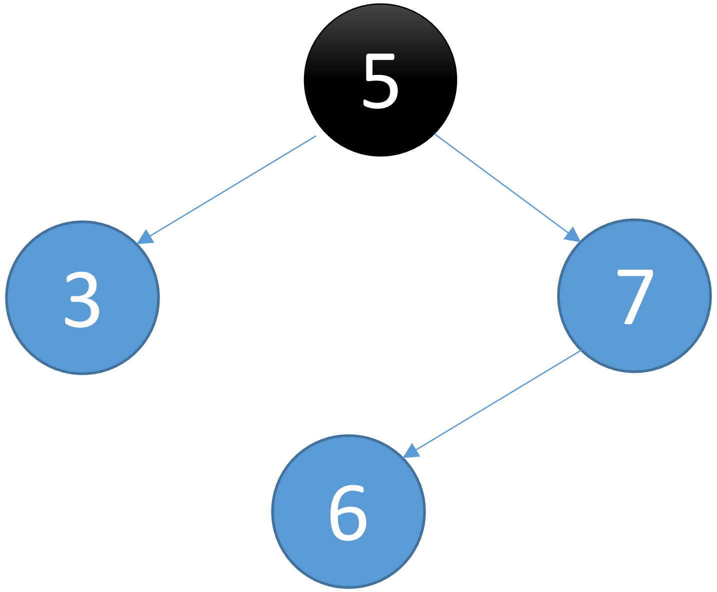
1. Each node must be labeled either red or black
2. The root must be labeled black
3. The tree cannot have two red nodes in a row (for any red node its parent, left, and right must be black)
4. Every root-NULL path must include the same number of black nodes

Red-Black Trees



1. Each node must be labeled either red or black
2. The root must be labeled black
3. The tree cannot have two red nodes in a row (for any red node its parent, left, and right must be black)
4. Every root-NULL path must include the same number of black nodes

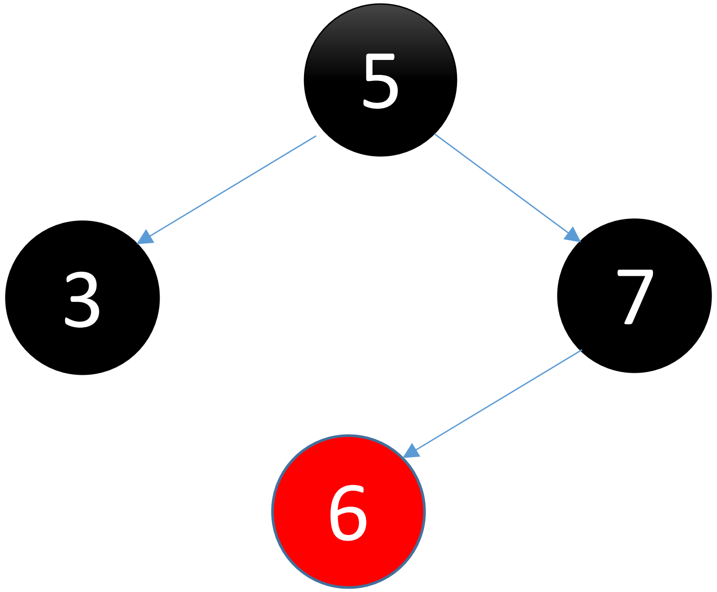
Red-Black Trees



1. Each node must be labeled either red or black
2. The root must be labeled black
3. The tree cannot have two red nodes in a row (for any red node its parent, left, and right must be black)
4. Every root-NULL path must include the same number of black nodes

Color this as a Red-Black Tree

Red-Black Trees



1. Each node must be labeled either red or black
2. The root must be labeled black
3. The tree cannot have two red nodes in a row (for any red node its parent, left, and right must be black)
4. Every root-NULL path must include the same number of black nodes

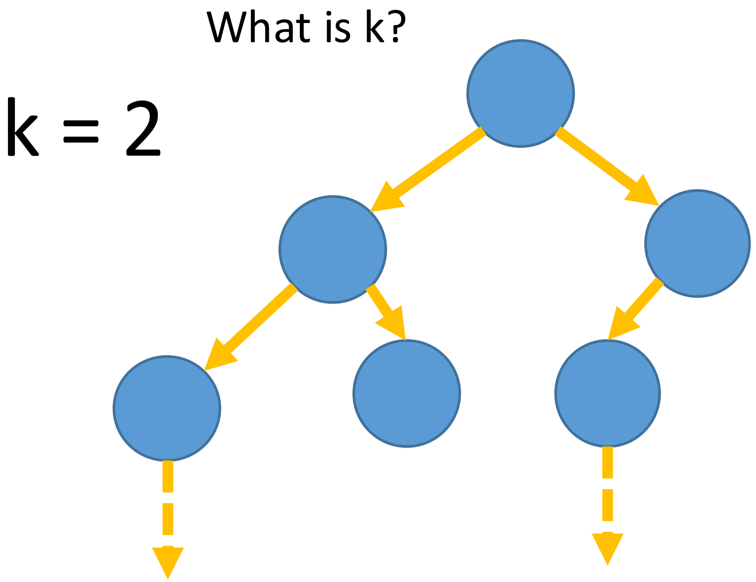
How did Red-Black Trees get their name?

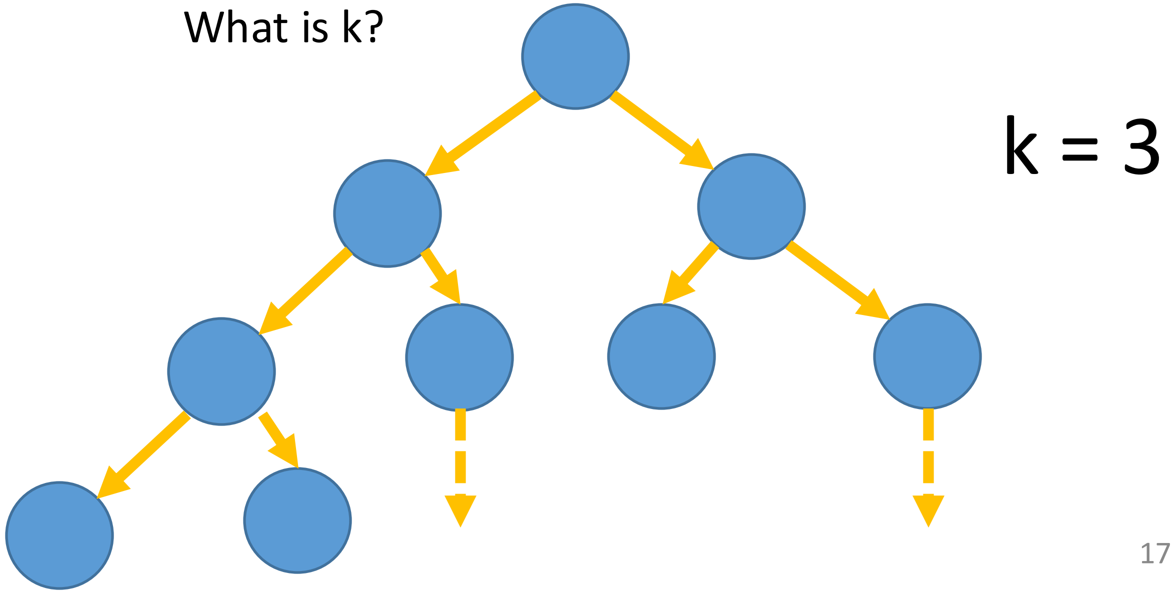
Plan

1. Prove the height property of a Red-Black tree.
2. Look at the insertion operation

Red-Black Tree Height

- **Claim**: every Red-Black tree has a $t_{height} \leq 2\lg(n+1) = O(\lg n)$
- Observation: if every root-NULL path has ≥ k nodes, then the tree includes a perfectly balanced (complete) top portion with k levels





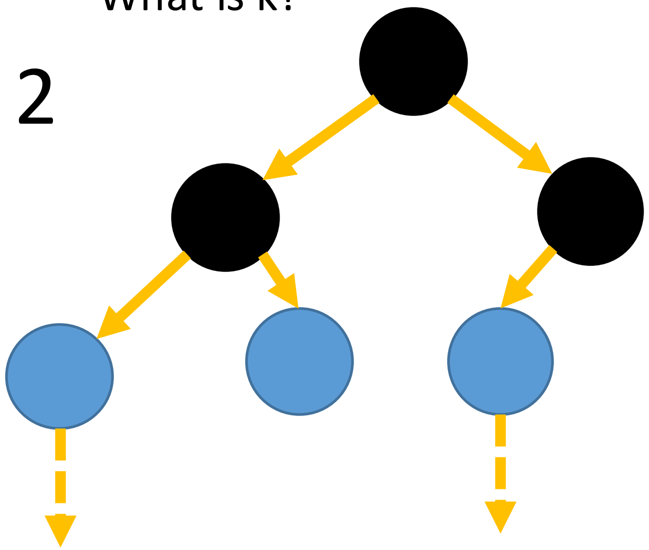
Red-Black Tree Height

- **Claim**: every Red-Black tree has a $t_{height} \leq 2\lg(n+1)$

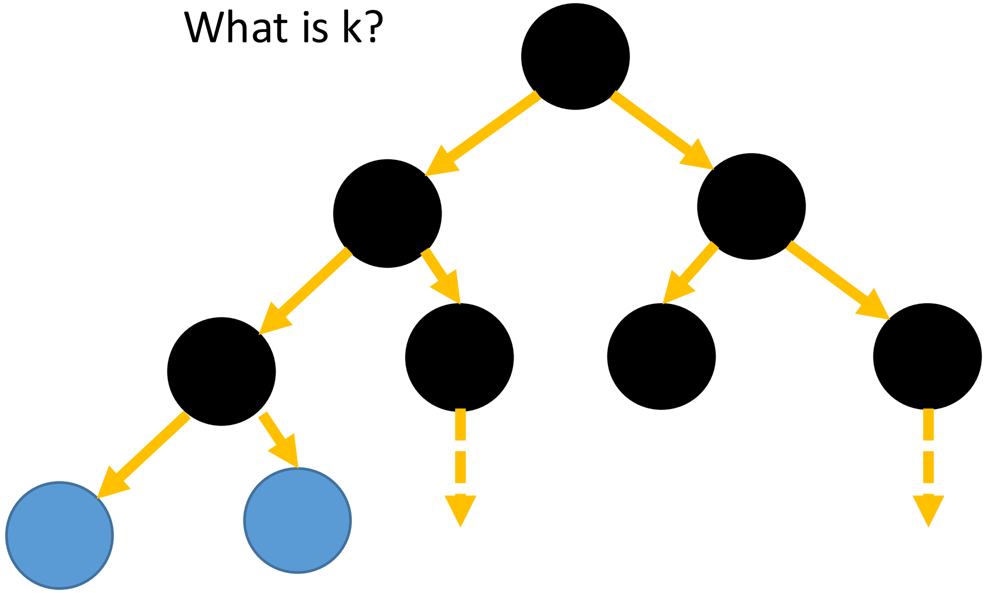
- Observation: if every root-NULL path has ≥ k nodes, then the tree includes a perfectly balanced (complete) top portion with k levels

What is k?

k = 2



What is k?



k = 3

k	n

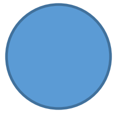
Red-Black Tree Height

- **Claim**: every Red-Black tree has a $t_{height} \leq 2\lg(n+1)$

- Observation: if every root-NULL path has ≥ k nodes, then the tree includes a perfectly balanced (complete) top portion with k levels

What is the minimum number of nodes (n) in the tree based on k?

Exercise question 1

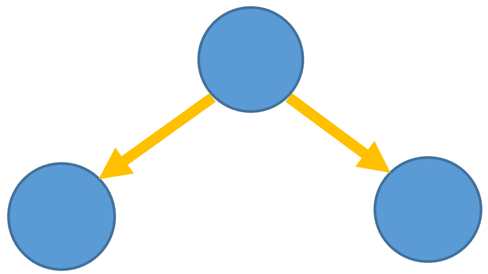


Red-Black Tree Height

k	n
1	1
2	

- **Claim**: every Red-Black tree has a $t_{height} \leq 2\lg(n+1)$
- Observation: if every root-NULL path has ≥ k nodes, then the tree includes a perfectly balanced (complete) top portion with k levels

What is the minimum number of nodes (n) in the tree based on k?

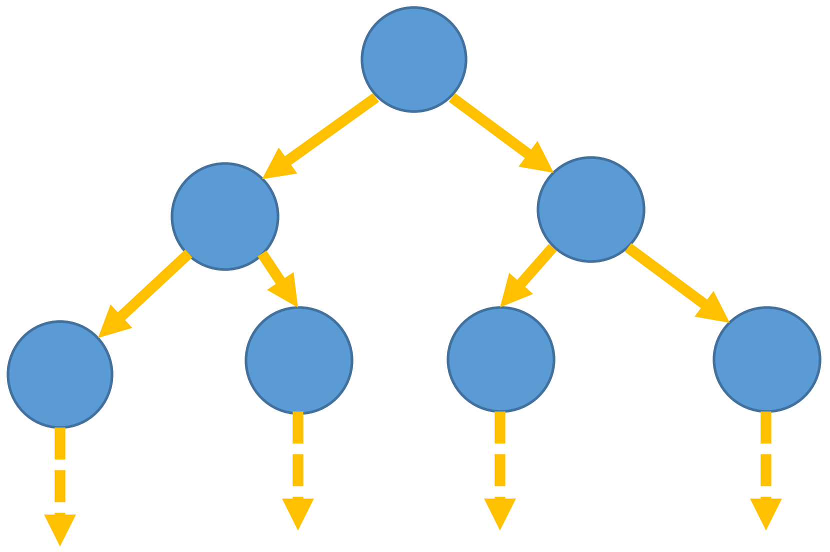


Red-Black Tree Height

k	n
1	1
2	3
3	
4	
5	
6	

- **Claim**: every Red-Black tree has a $t_{height} \leq 2\lg(n+1)$
- Observation: if every root-NULL path has ≥ k nodes, then the tree includes a perfectly balanced (complete) top portion with k levels

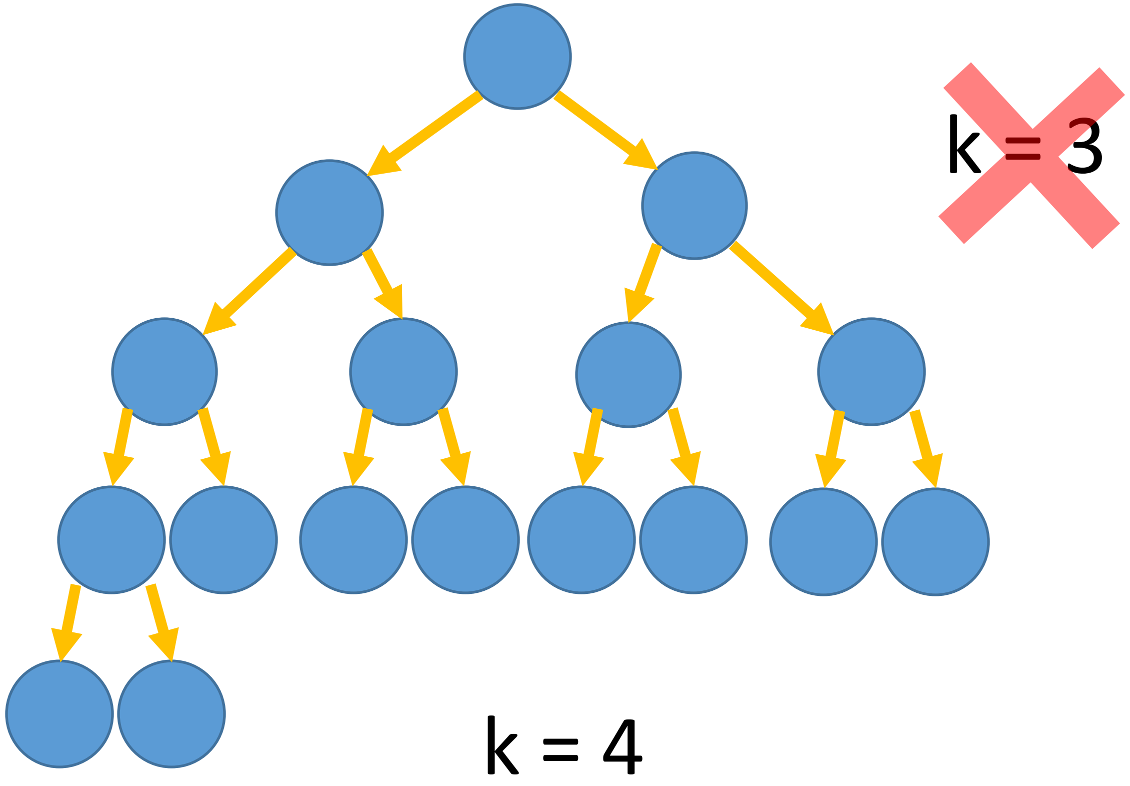
What is the minimum number of nodes (n) in the tree based on k?



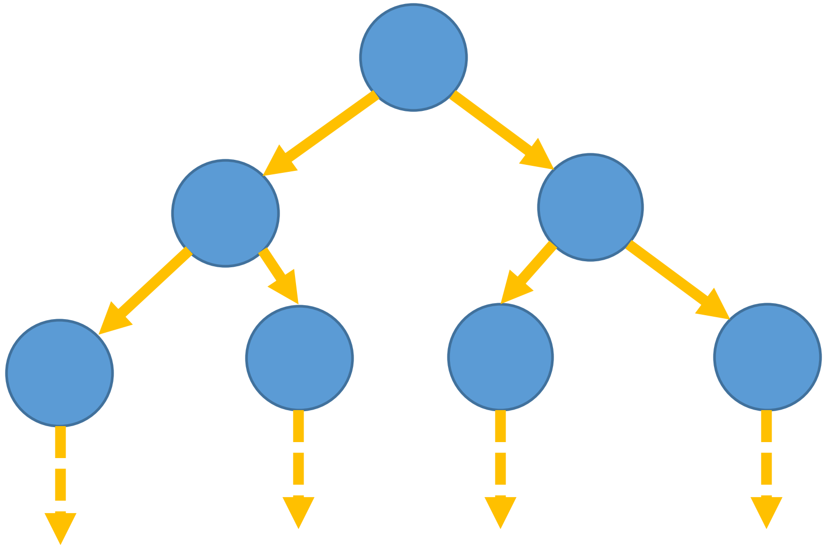
Red-Black Tree Height

- **Claim**: every Red-Black tree has a $t_{height} \leq 2\lg(n+1)$
- Observation: if every root-NULL path has ≥ k nodes, then the tree includes a perfectly balanced (complete) top portion with k levels

What is the minimum number of nodes (n) in the tree based on k?



Red-Black Tree Height



2^k - 1 was the minimum number of nodes

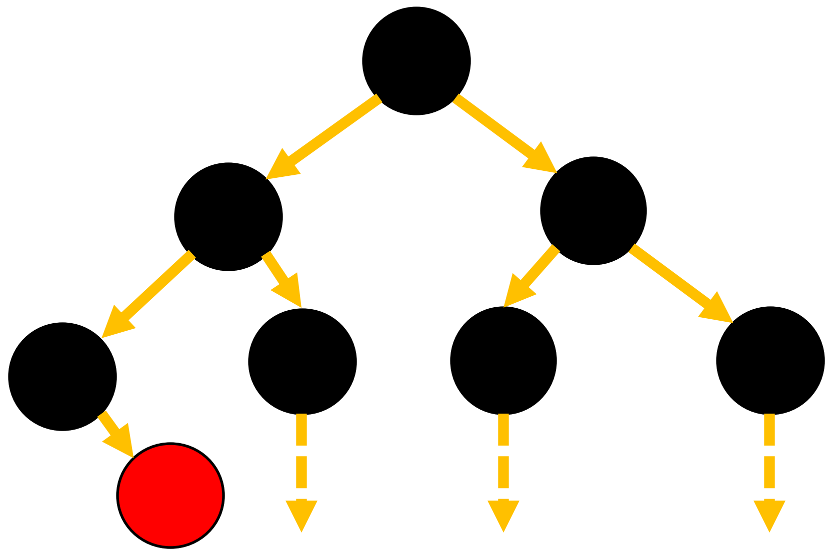
- So, we have:

$$n \geq 2^k - 1$$
$$\lg(n+1) \geq k$$

- So, we now have an upper bound on k.
- But how does k help us bound the actual height of the tree?
- What does k tell us about the number of black nodes you can have?
- What is the maximum number of black nodes on any root-Null path?

Observation: if every root-NULL path has ≥ k nodes, then the tree includes a perfectly balanced top portion with k levels

Red-Black Tree Height



- So, we have:

$$n \geq 2^k - 1$$
$$\lg(n+1) \geq k$$

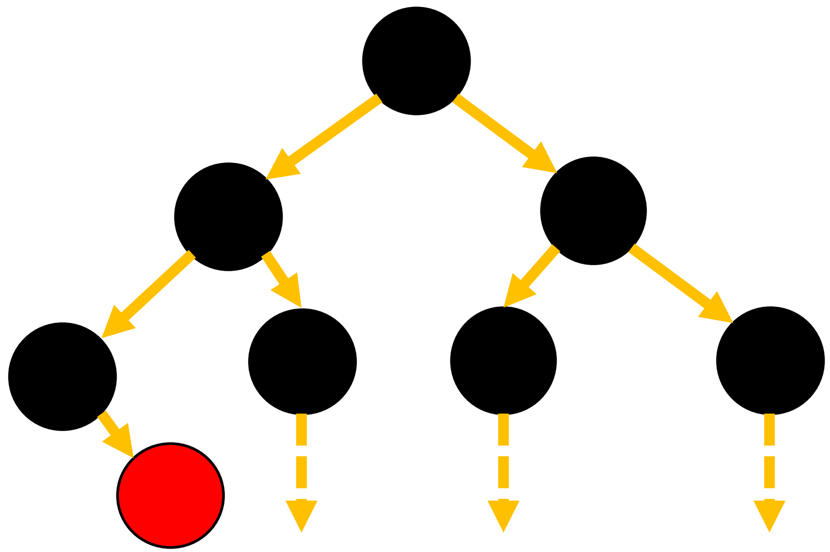
- So, we now have an upper bound on k.
- But how does k help us bound the actual height of the tree?
- What does k tell us about the number of black nodes you can have?
- What is the maximum number of black nodes on any root-Null path?

Observation: if every root-NULL path has ≥ k nodes, then the tree includes a perfectly balanced top portion with k levels

At most k black nodes

At most lg(n + 1) black nodes

Red-Black Tree Height



- So, we have:

$$n \geq 2^k - 1$$
$$\lg(n+1) \geq k$$

- So,
- But … height of the tree?
- What does k tell us about the number of black nodes you can have?
- What is the maximum number of black nodes on any root-Null path?

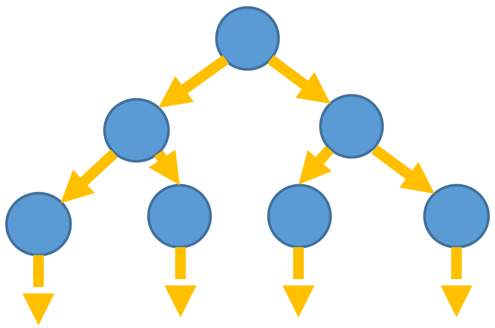
How many **red** nodes on any root-Null path?

Observation: if every root-NULL path has ≥ k nodes, then the tree includes a perfectly balanced top portion with k levels

At most k black nodes

At most lg(n + 1) black nodes

Red-Black Tree Height



- Thus: in a Red-Black tree with n nodes, there is a root-NULL path with **at most** lg (n + 1) **black** nodes
- By invariant (4): every root-NULL path has $\leq \lg(n+1)$ **black** nodes
- By invariant (3): every root-NULL path has $\leq \lg(n+1)$ **red** nodes
- Thus, a **total** of $\leq 2\lg(n+1)$ nodes on every root-NULL path

1. Each node must be labeled either red or black
2. The root must be labeled black
3. The tree cannot have two red nodes in a row (for any red node its parent, left, and right must be black)
4. Every root-NULL path must include the same number of black nodes

Red-Black Trees

- If our tree can be colored as a Red-Black tree, then every root-NULL path has $\leq 2\lg(n+1)$ nodes total

- The longest path will dictate the height of the tree

- So, height of the tree is at most $2\lg(n+1)$

  lg(n+1) = lg n + lg(1 + 1/n) = lg n + C

- A tree cannot contain a *chain* of three nodes

- Thus, the height of the tree is O(lg n)
- Why is this important?

Draw a Worst-Case (most lopsided) Red-Black Tree with a **minimum** of 3 black nodes on every root-NULL path

Let’s look at

- Inserting into a Red-Black Tree
- Deleting nodes from a Red-Black Tree (probably left for outside class)

Red-Black Trees, Inserting a Node

1. Insert the new node

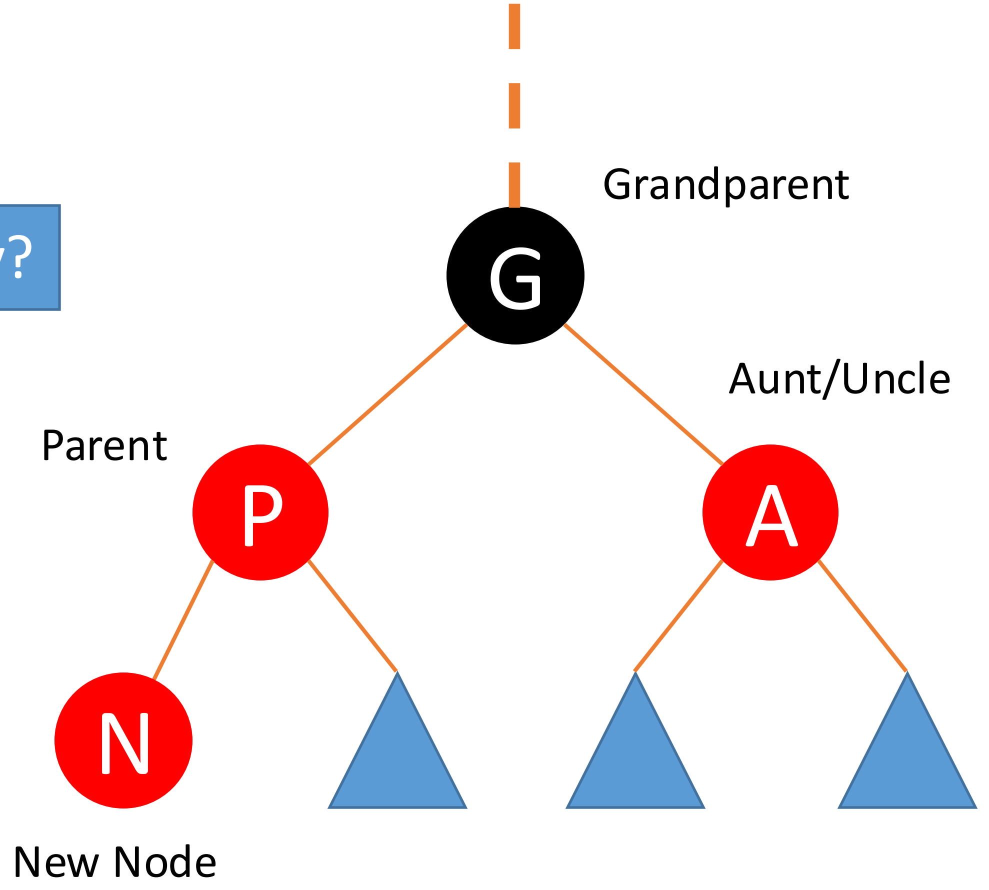
2. Color it red

3. Fix colors to enforce Red-Black Tree invariants
   - This is a recursive process

Red-Black Trees, Inserting a Node

1. Insert the new node (always insert as a leaf) **Why?**
2. If the inserted node is the root, then color it black, otherwise color it red **Why?**
3. If the new node is not root and its parent is black, then we are done
4. Otherwise, look at the node's aunt
   a) If aunt is red
      I. Change color of parent and aunt to black
      II. Change color of the new node and the grandparent to red
      III. Go to step (2) and treat grandparent as new node

Move the black color down

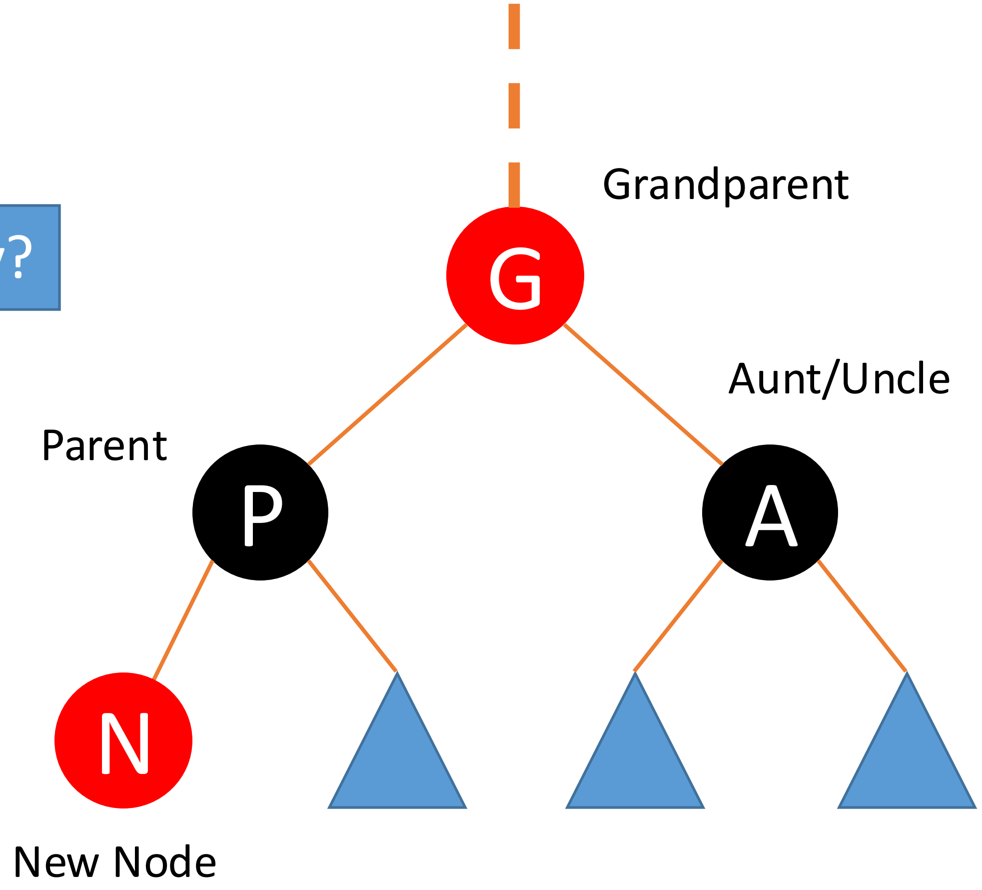


"Aunt" is usually called "Uncle"

Red-Black Trees, Inserting a Node

1. Insert the new node (always insert as a leaf) **Why?**
2. If the inserted node is the root, then color it black, otherwise color it red **Why?**
3. If the new node is not root and its parent is black, then we are done
4. Otherwise, look at the node's aunt
   a) If aunt is red
      I. Change color of parent and aunt to black
      II. Change color of the new node and the grandparent to red
      III. Go to step (2) and treat grandparent as new node

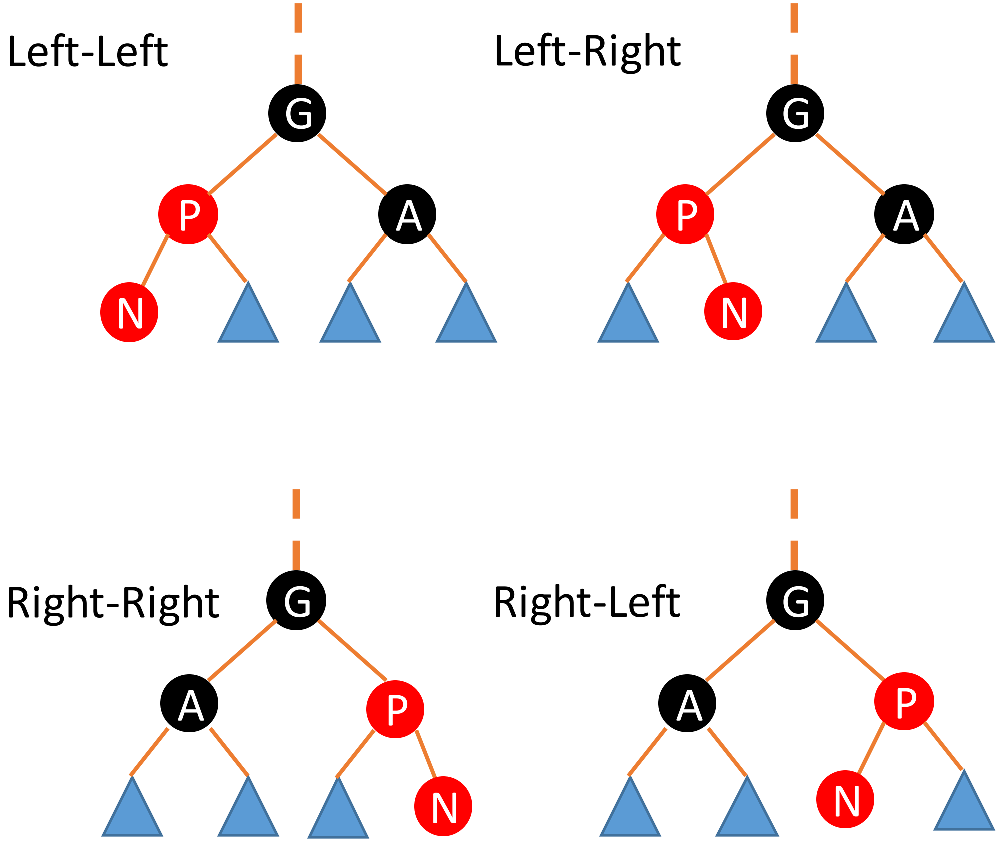
Move the black color down



"Aunt" is usually called "Uncle"

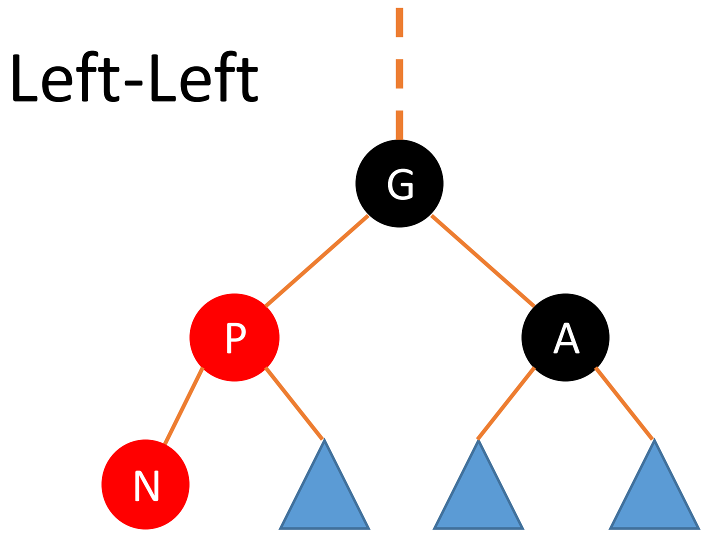
Red-Black Trees, Inserting a Node

1. Insert the new node
   (always insert as a leaf)
2. If the inserted node is the root, then color it black, otherwise color it red
3. If the new node is not root and its parent is black, then we are done
4. Otherwise, look at the node's aunt
   a) If aunt is red
   b) If aunt is black (or does not exist)
      I. Put the new node, its parent, and the grandparent "in order" with the middle node as the root
      II. We have four possibilities for the current positions of N, P, and G

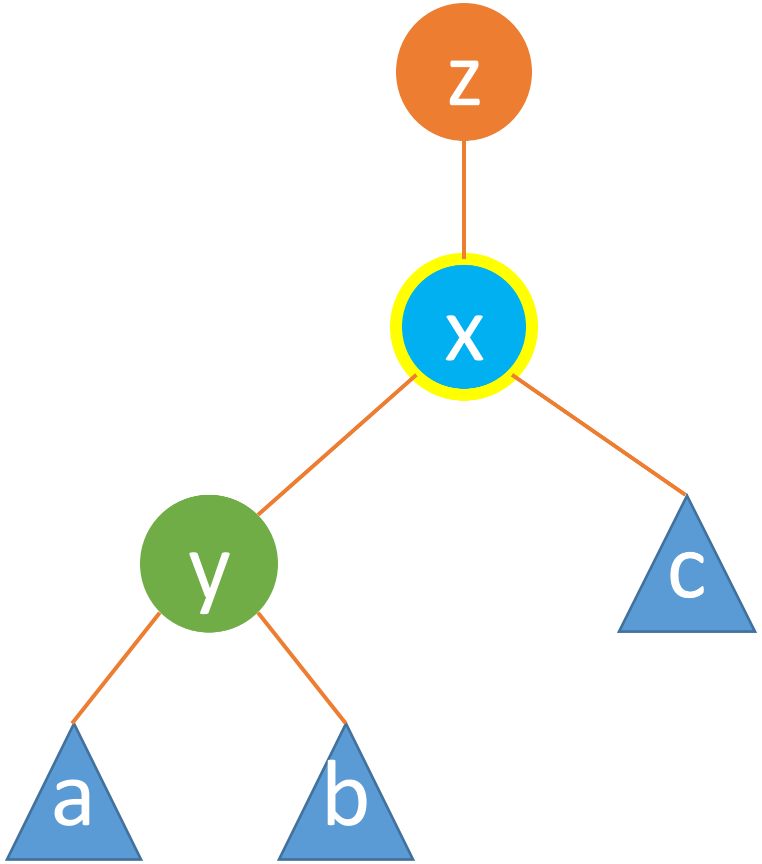


Red-Black Trees, Inserting a Node: Left-Left

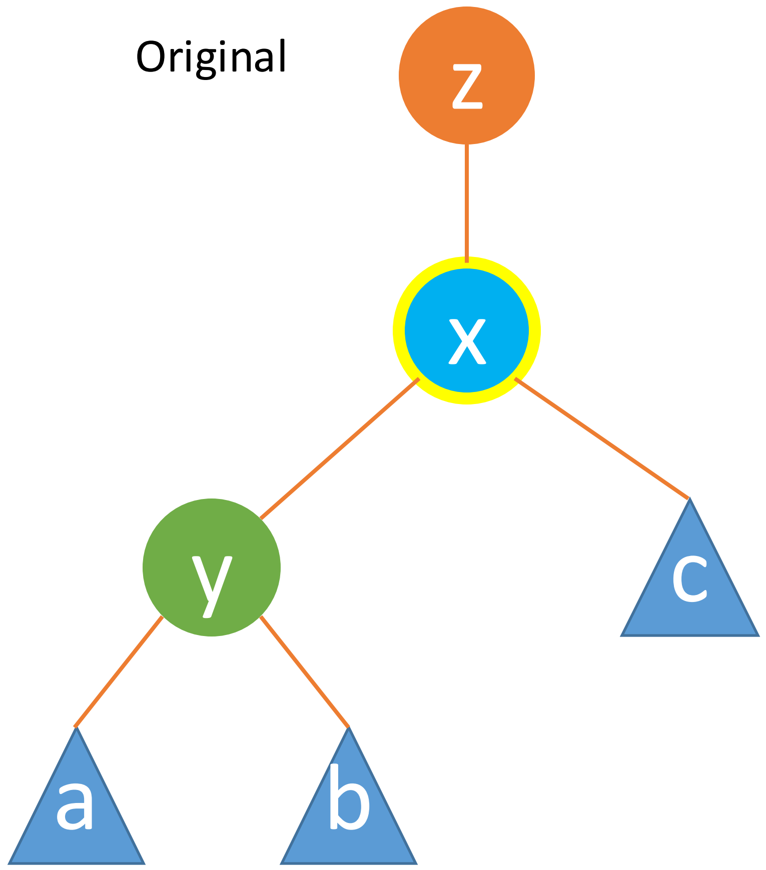
1. Right rotate around the grandparent

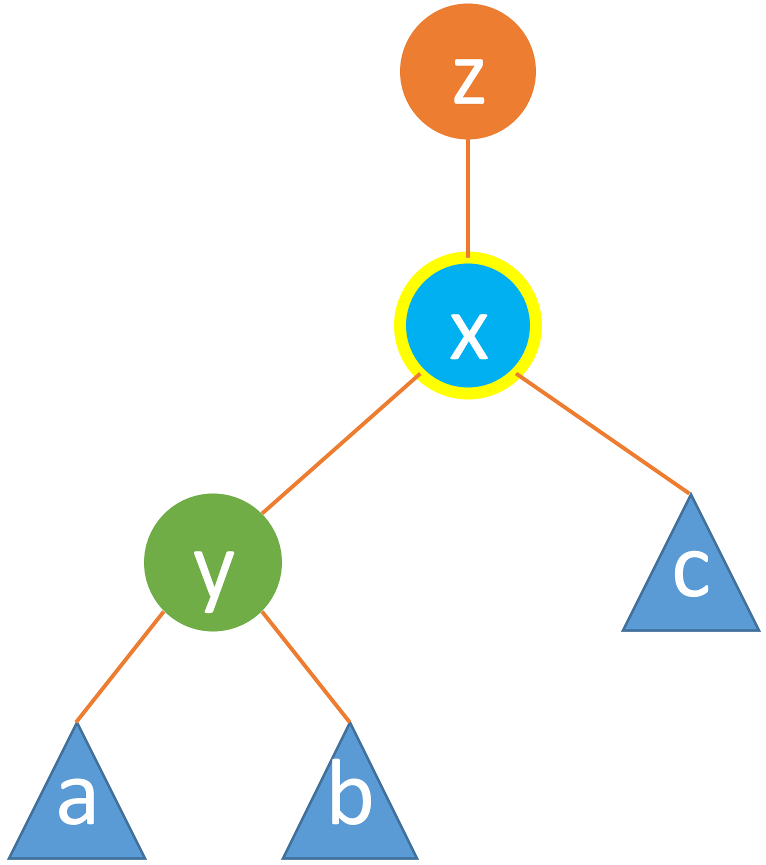


Tree Rotations: Right

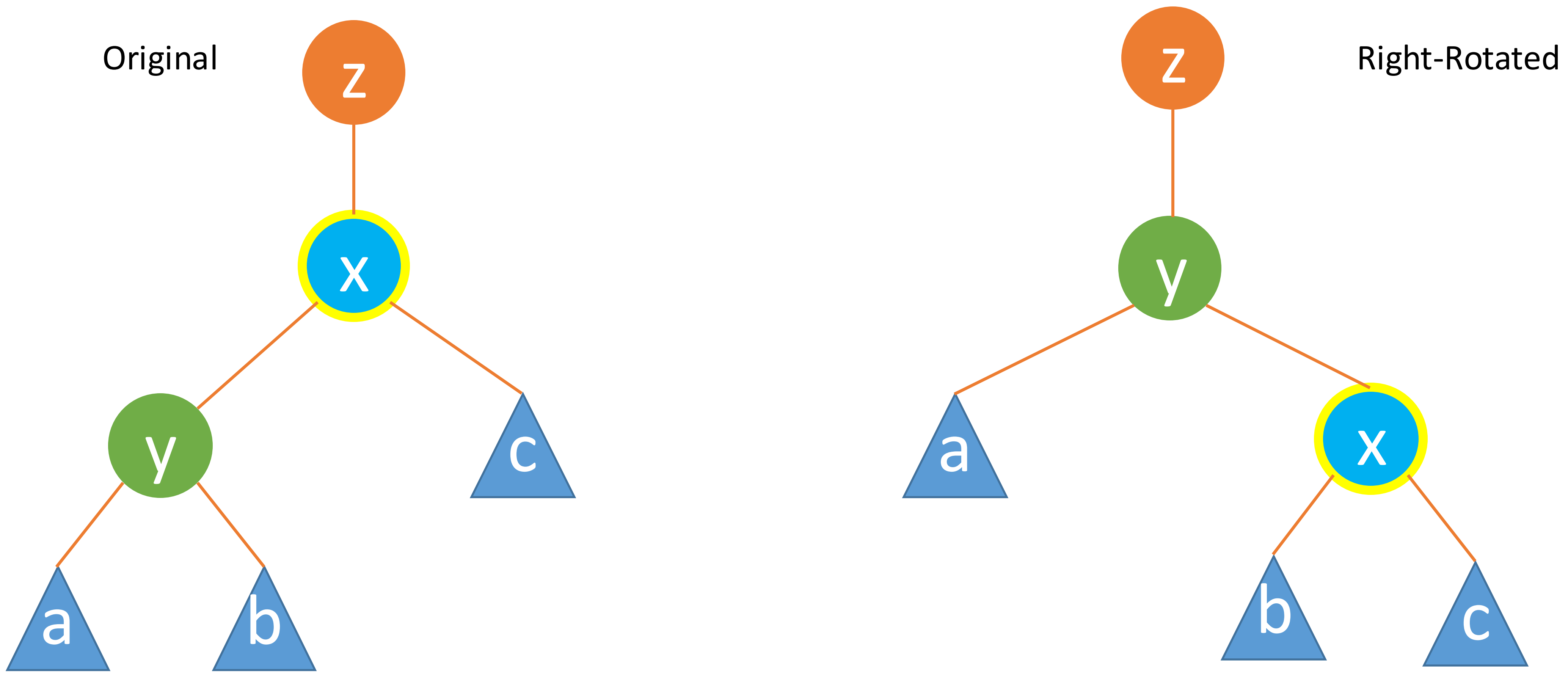


Tree Rotations: Right

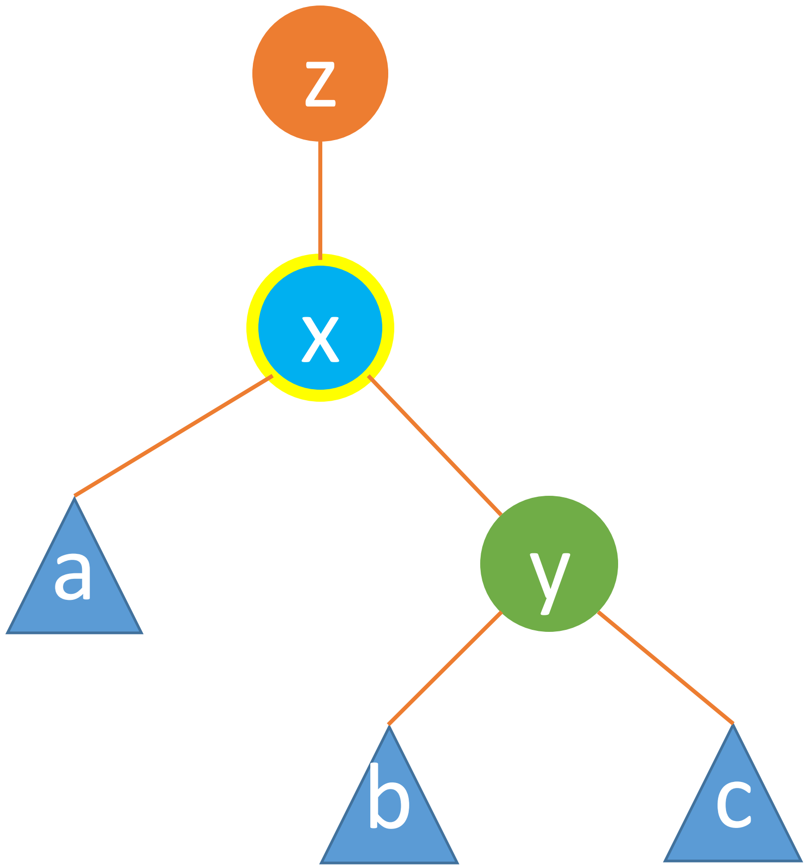




Tree Rotations: Right

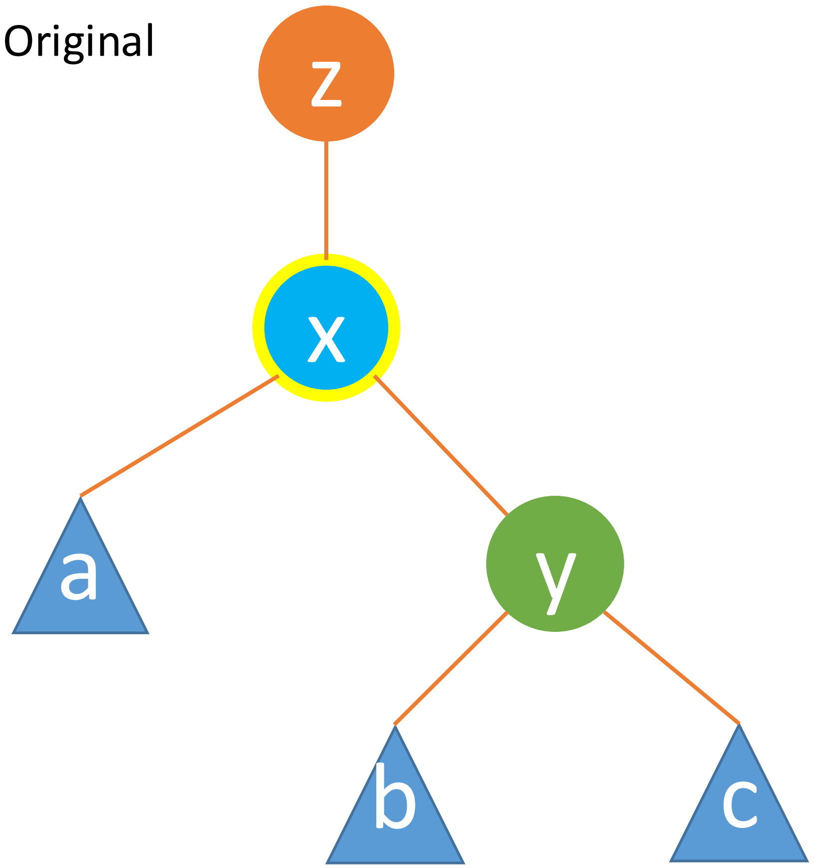


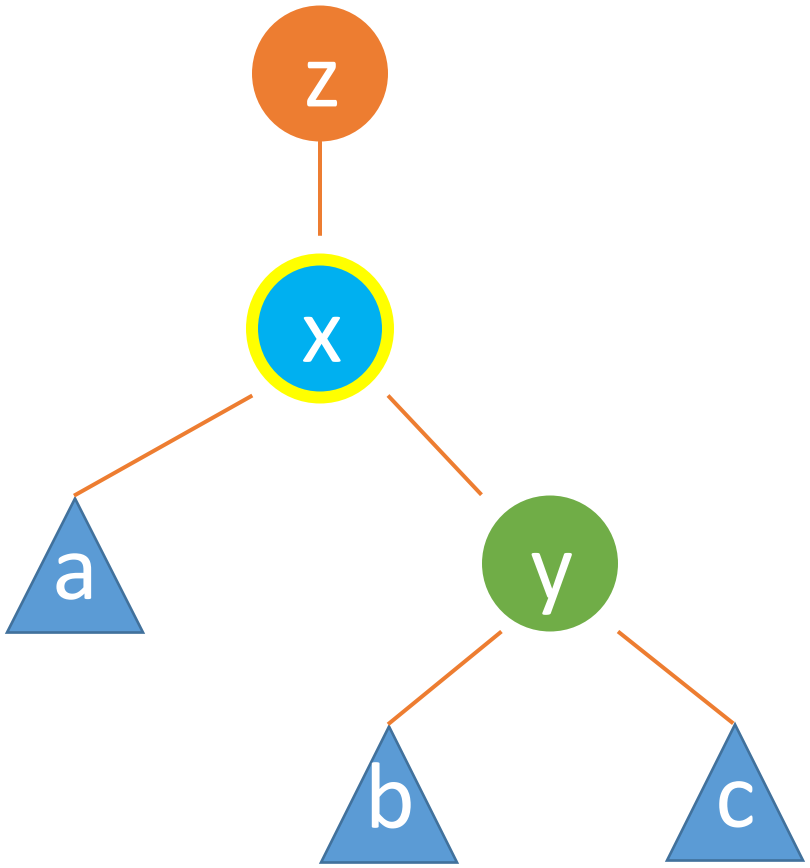
Tree Rotations: Left



Tree Rotations: Left

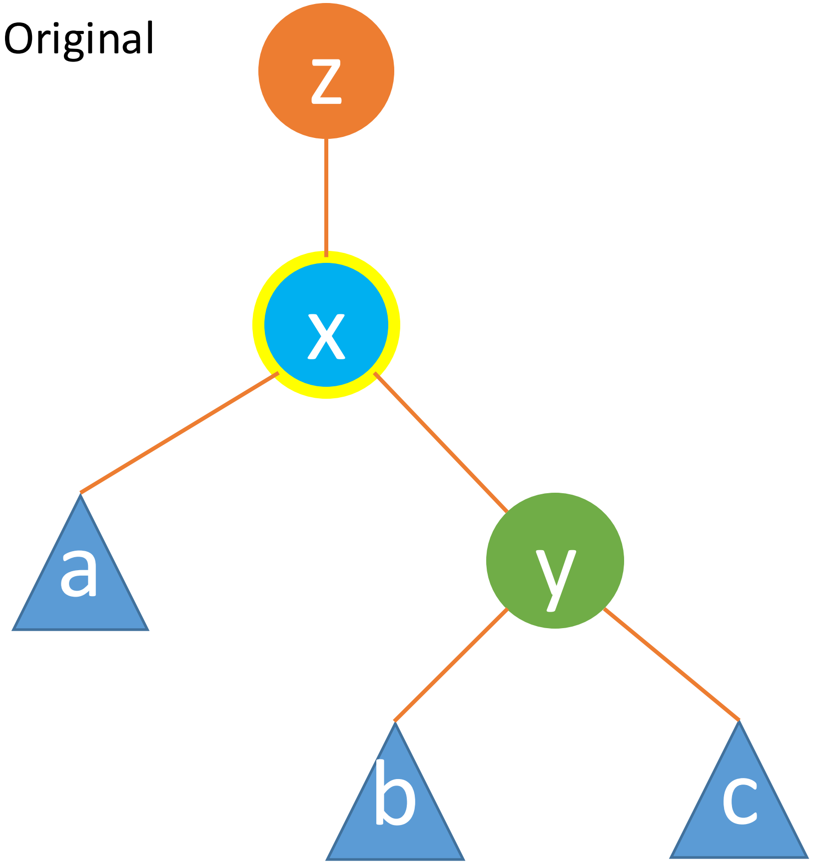
Original

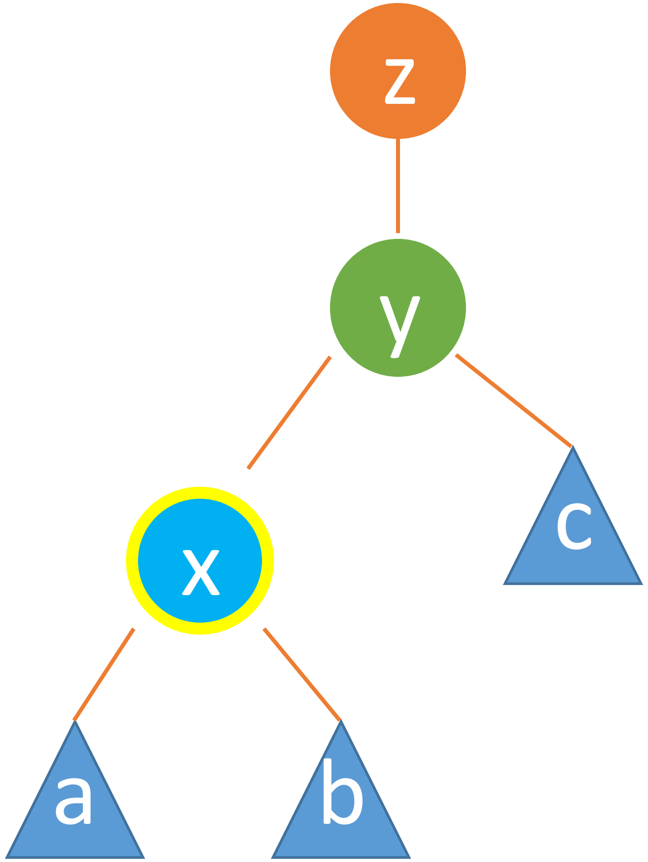




Tree Rotations: Left

Original

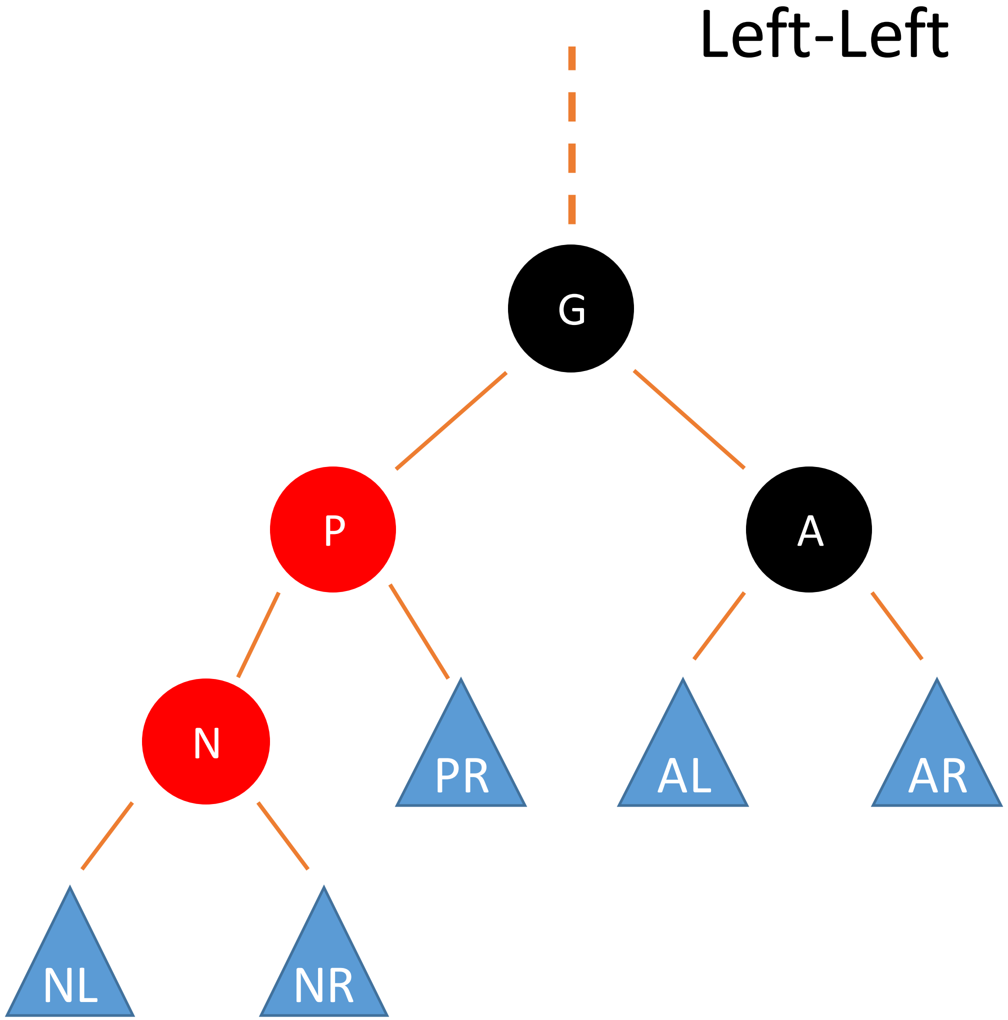




Left-Rotated

Red-Black Trees, Inserting a Node: Left, Left

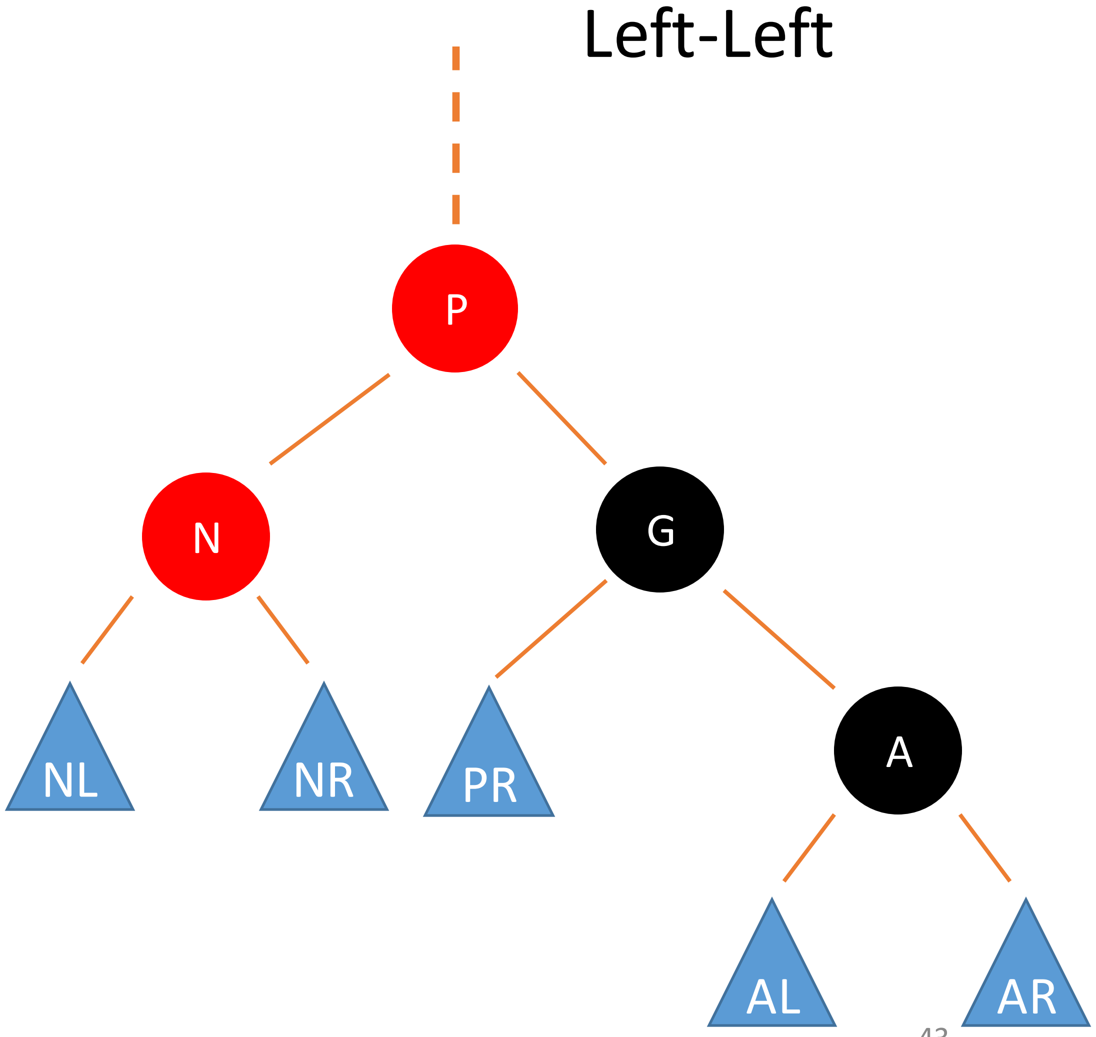
1. Right rotate around the grandparent



Red-Black Trees, Inserting a Node: Left, Left

1. Right rotate around the grandparent

2. Swap the colors of the grandparent and the parent

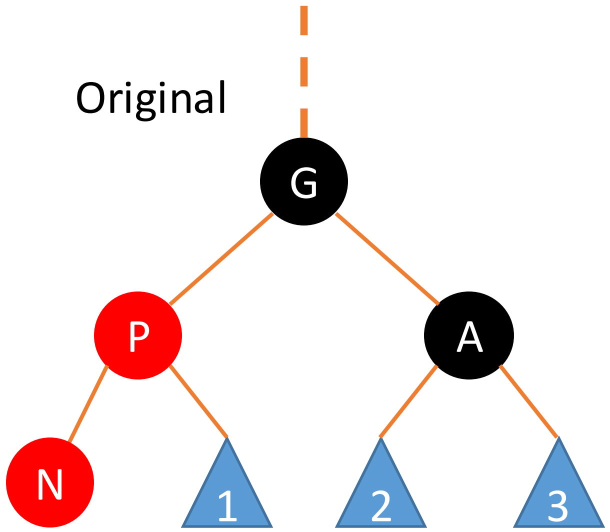


Red-Black Trees, Inserting a Node: Left, Left

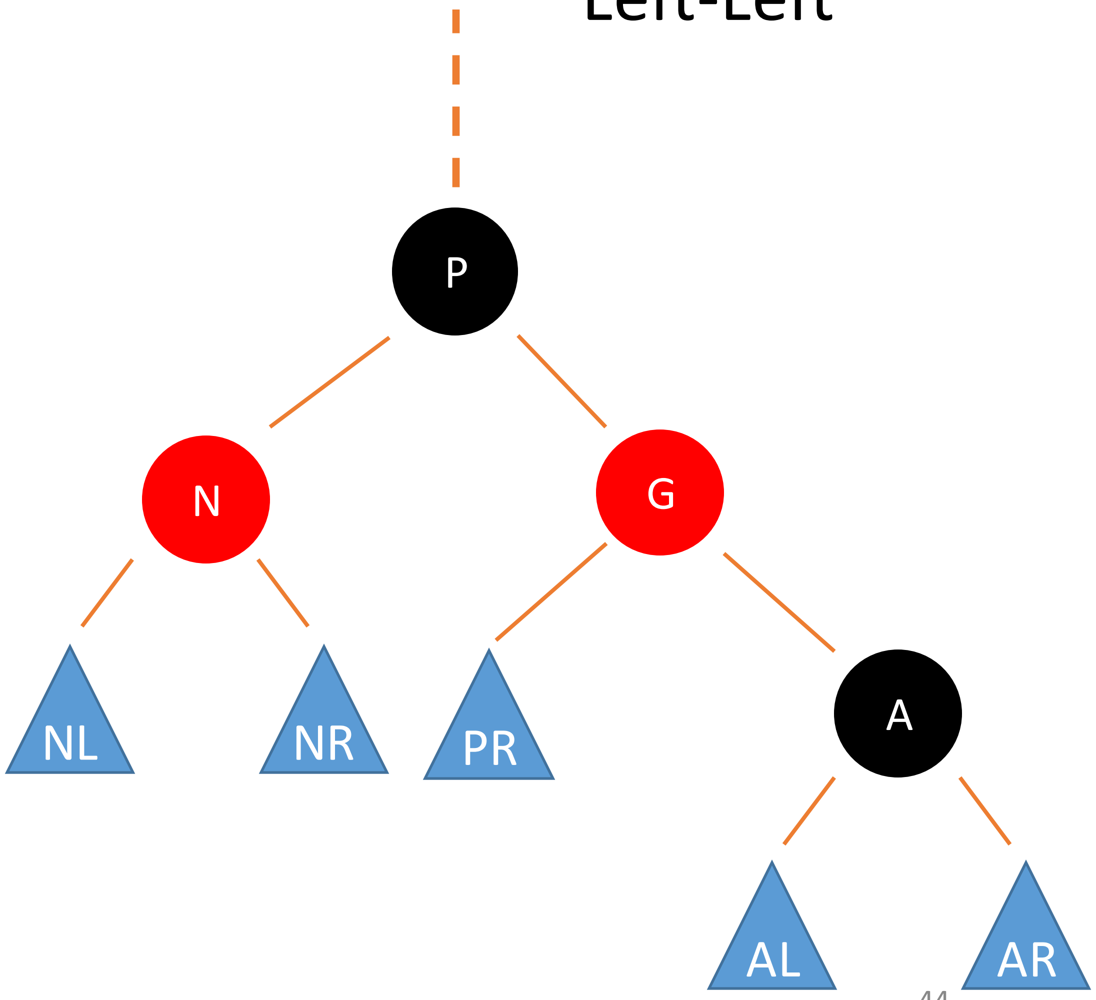
1. Right rotate around the grandparent

2. Swap the colors of the grandparent and the parent

Original



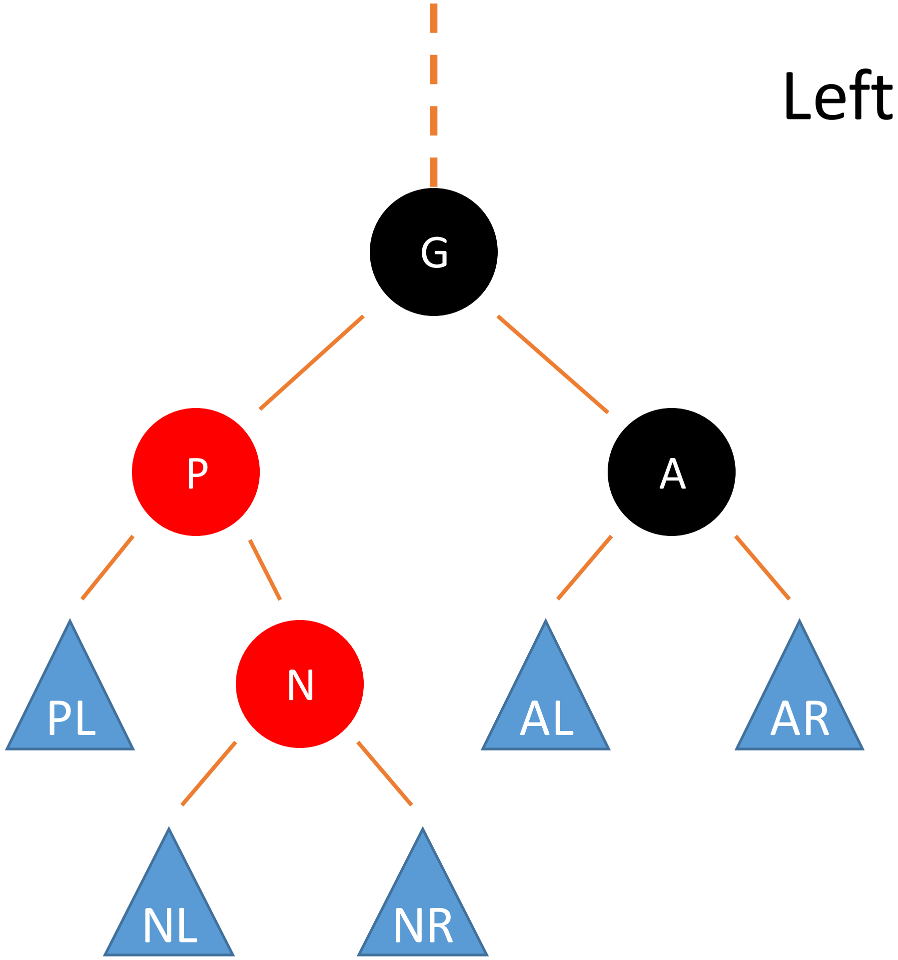
Left-Left



Red-Black Trees, Inserting a Node: Left, Right

1. Left rotate around the parent

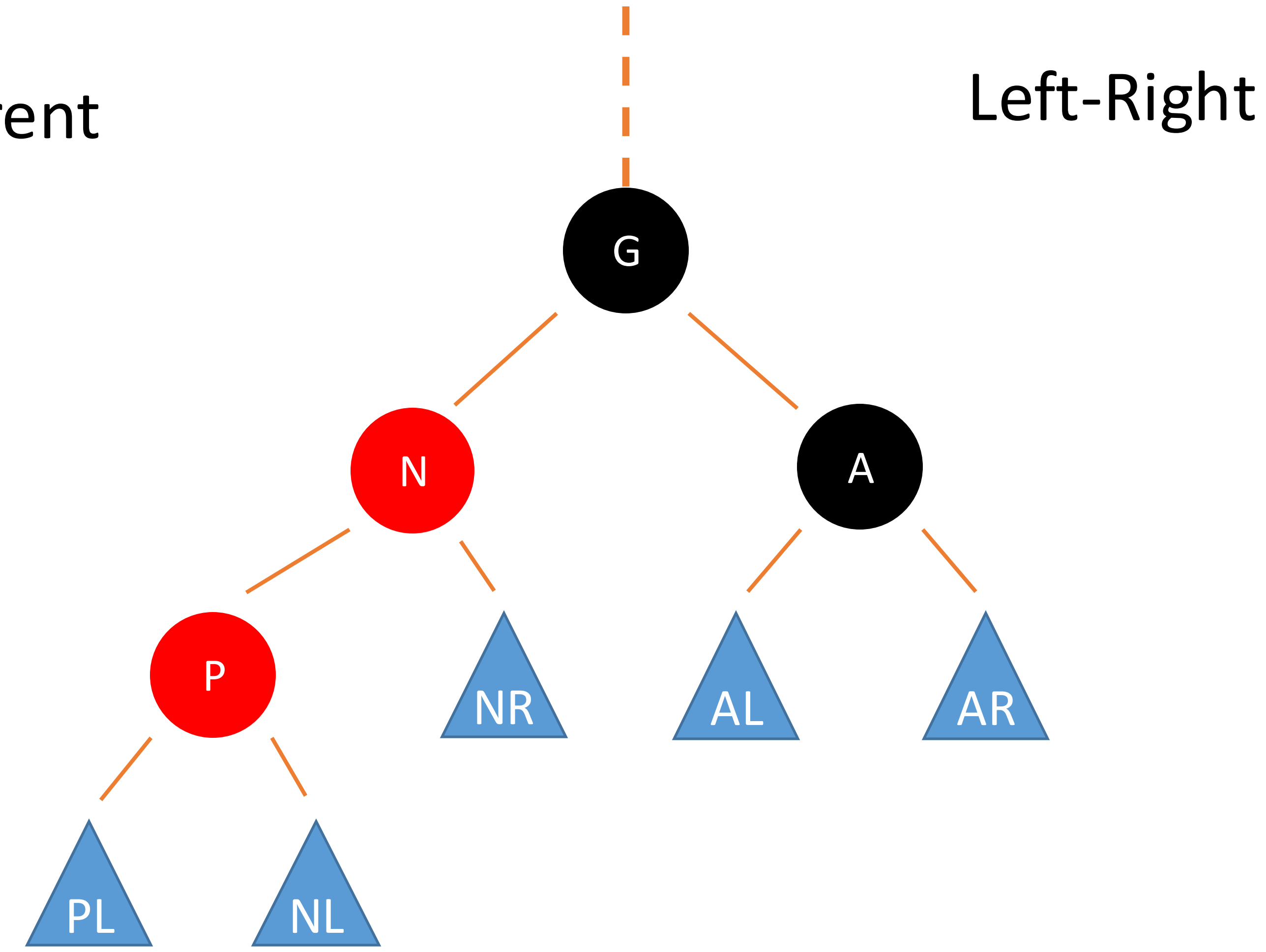
Left-Right



Red-Black Trees, Inserting a Node: Left, Right

1. Left rotate around the parent

2. Right rotate around the grandparent



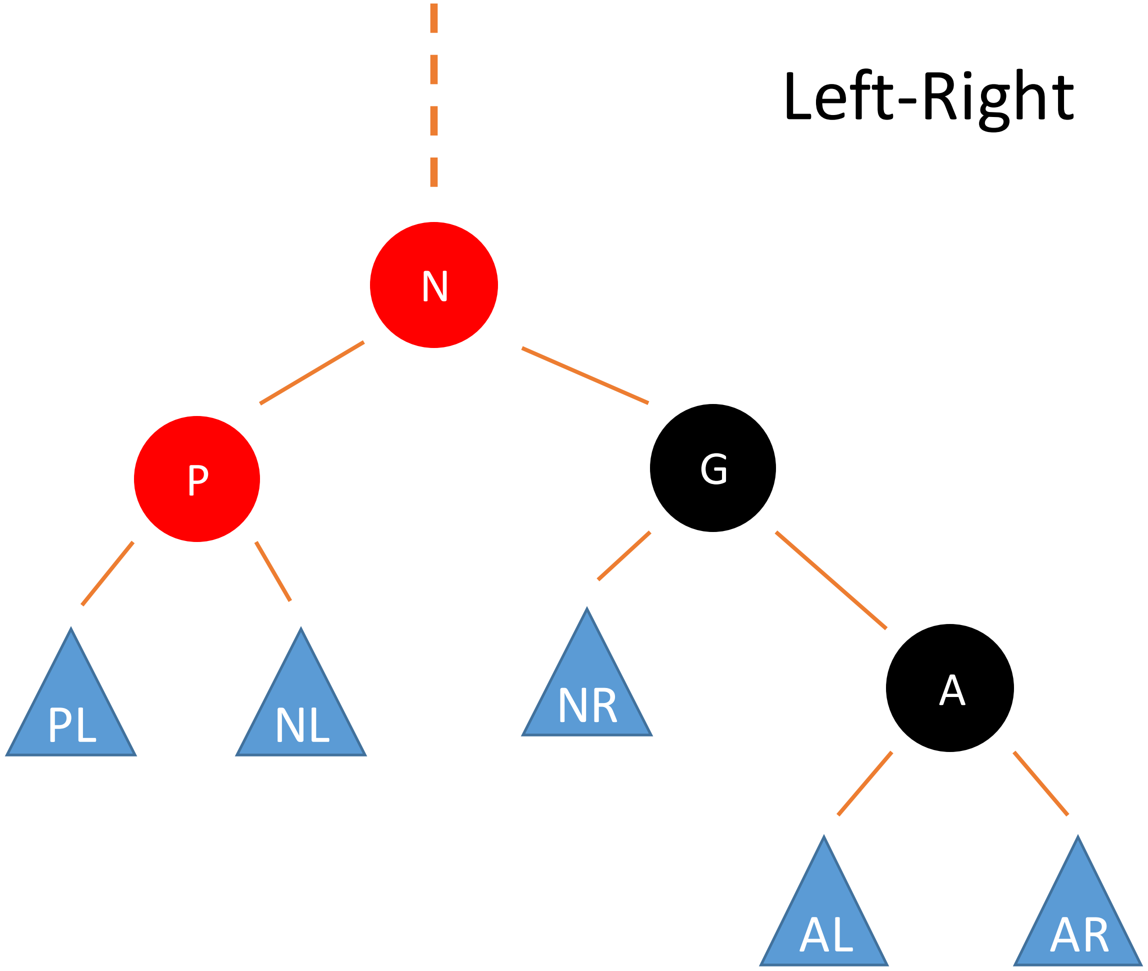
Red-Black Trees, Inserting a Node: Left, Right

1. Left rotate around the parent

2. Right rotate around the grandparent

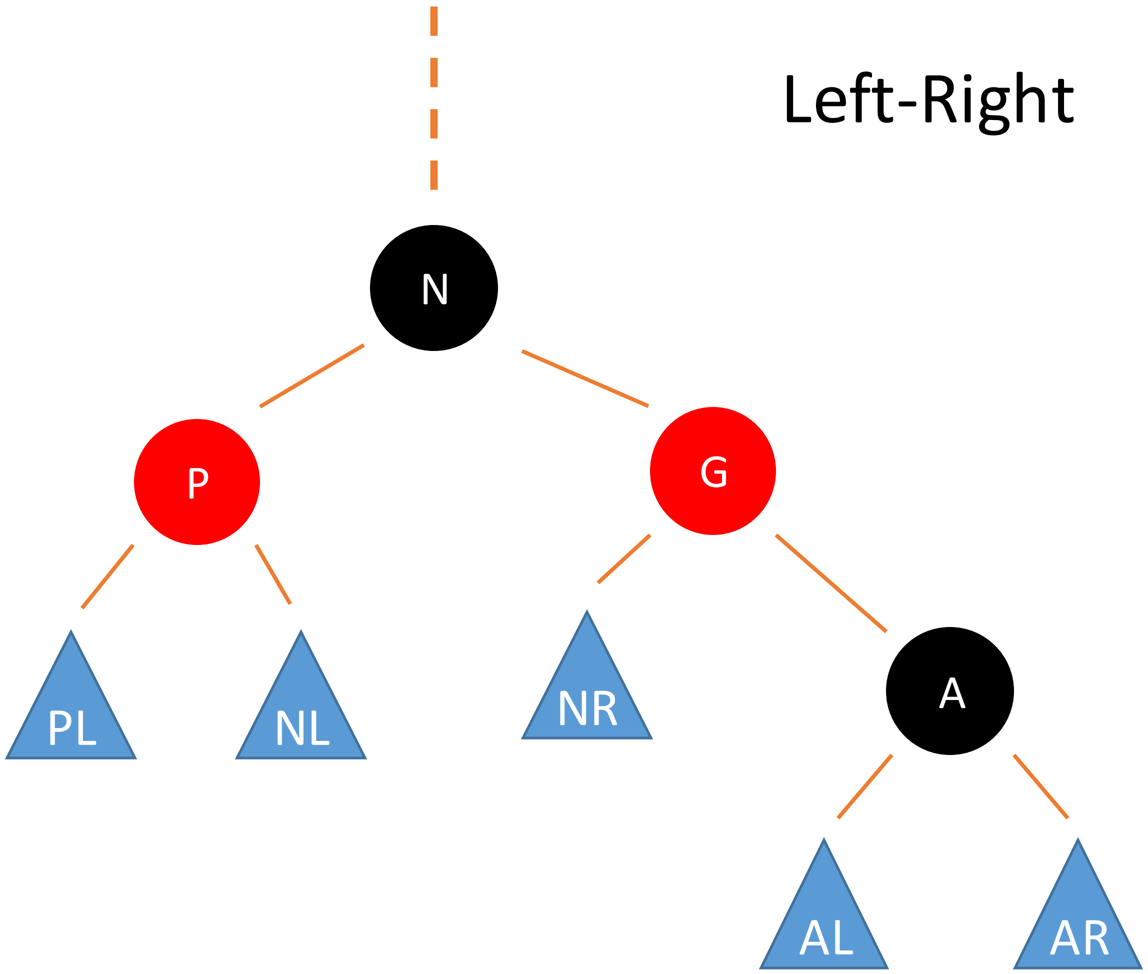
3. Swap the colors of the grandparent and the new node

Left-Right



Red-Black Trees, Inserting a Node: Left, Right

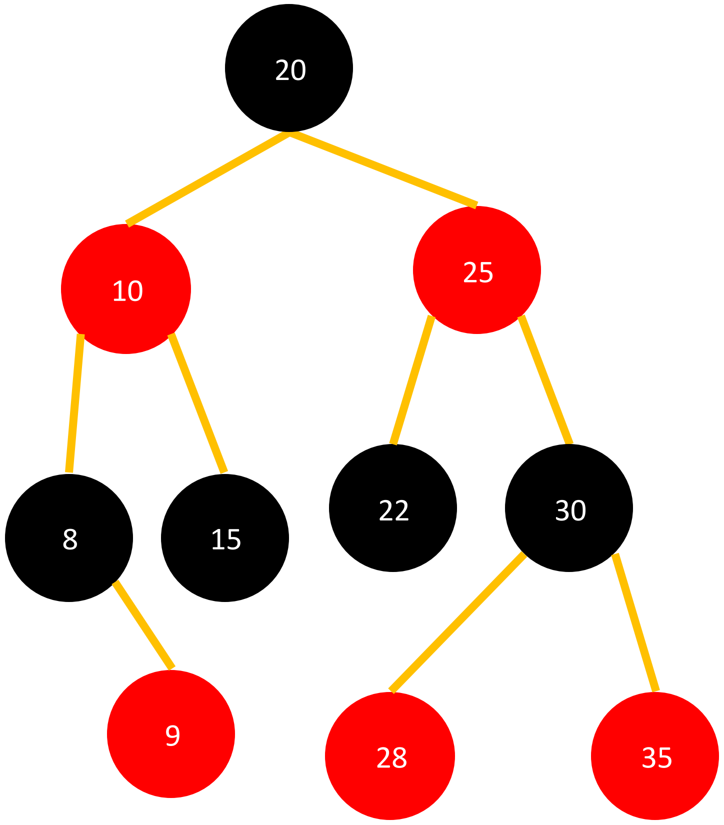
1. Left rotate around the parent
2. Right rotate around the grandparent
3. Swap the colors of the grandparent and the new node



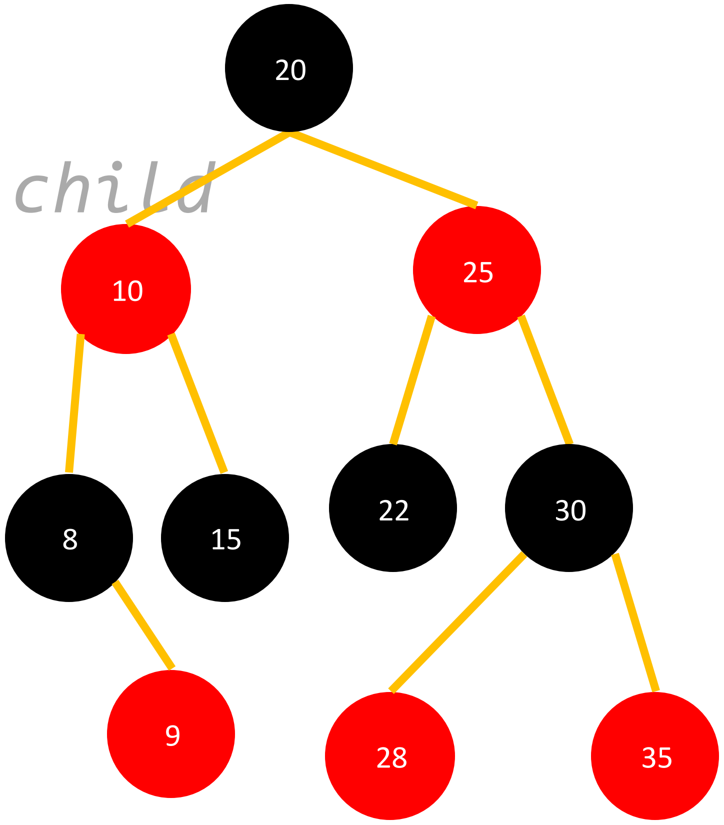
Red-Black Trees, Inserting a Node

- What about the Right-Right and Right-Left options?
- They are the inverse of the cases we’ve just covered.
- What are the running times of these procedures?
  - Inserting the new node?
  - Recoloring?
  - Restructuring?
- We’re not going to cover deletion, but what are your thoughts?
  - Operation? (http://www.geeksforgeeks.org/red-black-tree-set-3-delete-2/)
  - Running time?

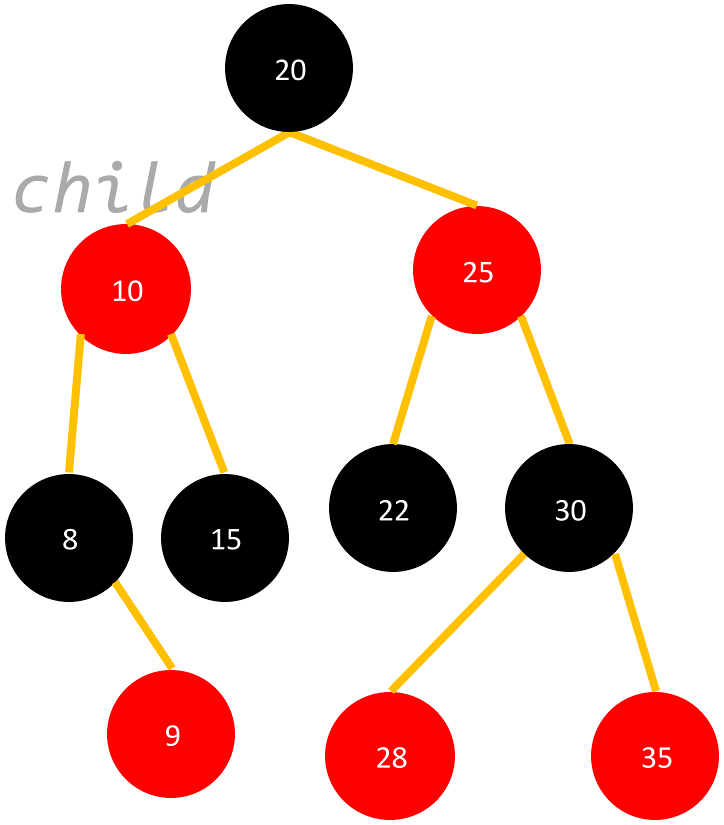
```
FUNCTION RBTreeInsert(tree, new_node)
    # Search for position of new_node
    parent = NONE
    current_node = tree.root
    WHILE current_node != NONE
        parent = current_node
        IF new_node.key < current_node.key
            current_node = current_node.left
        ELSE
            current_node = current_node.right
    new_node.parent = parent
```



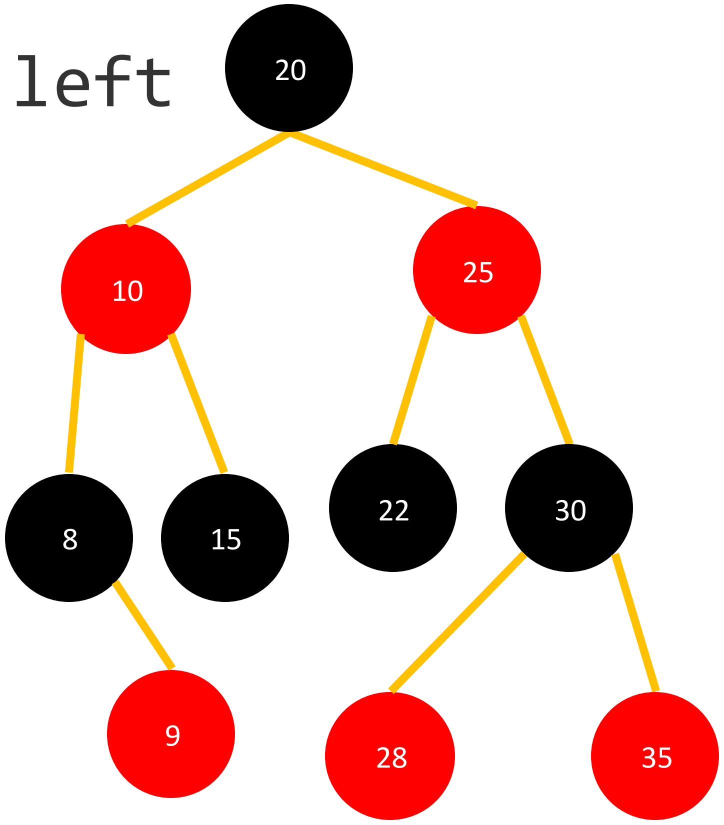
```
FUNCTION RBTreeInsert(tree, new_node)
    # Search for position of new_node
    …

    # Insert new_node as root or left/right child
    IF parent == NONE
        tree.root = new_node
    ELSE IF new_node.key < parent.key
        parent.left = new_node
    ELSE
        parent.right = new_node
```



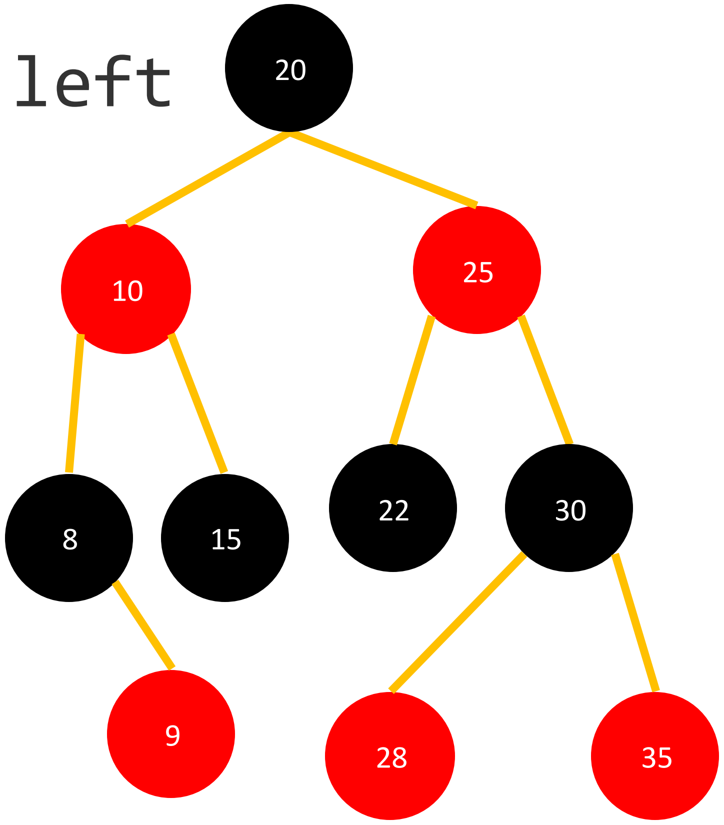
```
FUNCTION RBTreeInsert(tree, new_node)
    # Search for position of new_node
    …

    # Insert new_node as root or left/right child
    …

    # Initialize the new_node
    new_node.left = NONE
    new_node.right = NONE
    new_node.color = RED

    RBTreeFixColors(tree, new_node)
```



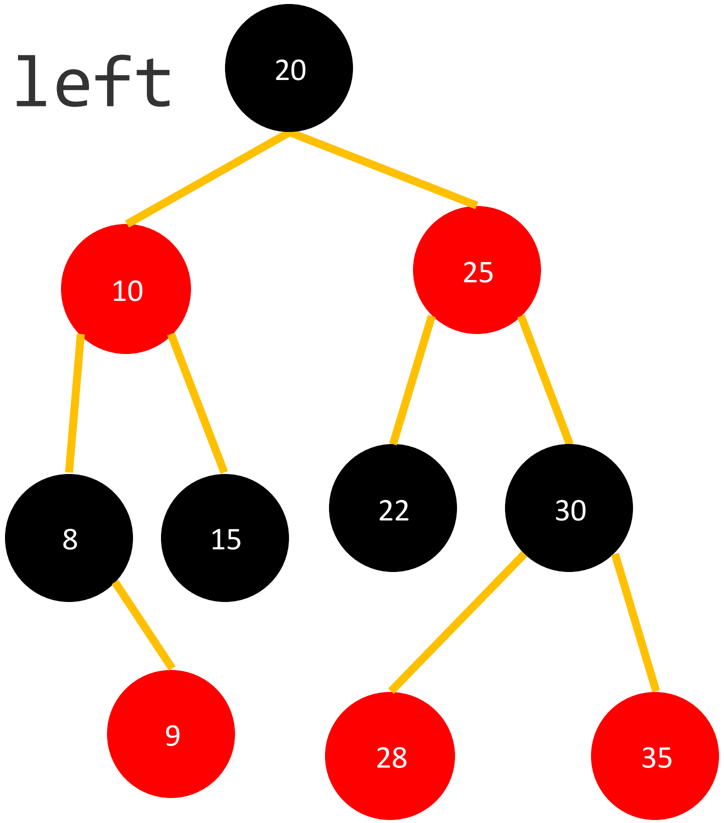
```
FUNCTION RBTreeFixColors(tree, node)
    WHILE node.parent.color == RED
        # Look for aunt/uncle node
        IF node.parent == node.parent.parent.left
            aunt = node.parent.parent.right
            IF aunt.color == RED
                node.parent.color = BLACK
                aunt.color = BLACK
                node.parent.parent.color = RED
                node = node.parent.parent
```



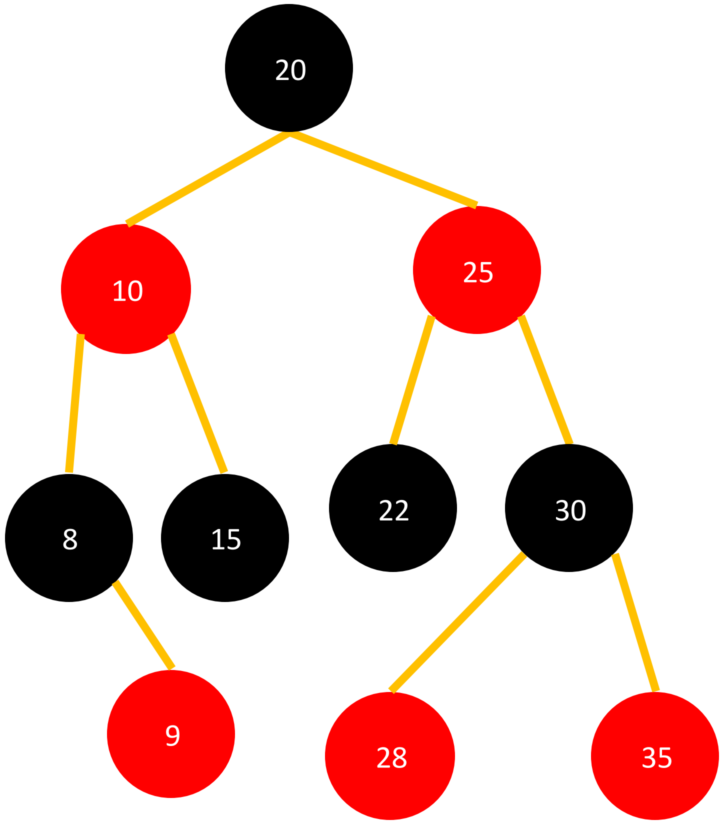
```
FUNCTION RBTreeFixColors(tree, node)
   WHILE node.parent.color == RED
      # Look for aunt/uncle node
      IF node.parent == node.parent.parent.left
         aunt = node.parent.parent.right
         IF aunt.color == RED
            …
         ELSE
           IF node == node.parent.right
              node = node.parent
              LeftRotate(tree, node)
           node.parent.color = BLACK
           node.parent.parent.color = RED
           RightRotate(tree, node.parent.parent)
```



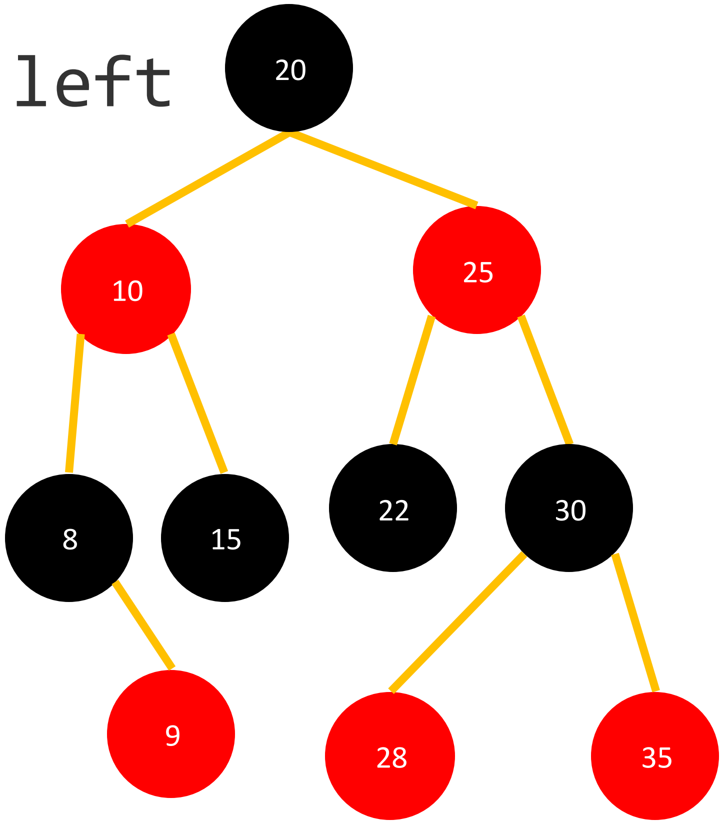
```
FUNCTION RBTreeFixColors(tree, node)
    WHILE node.parent.color == RED
        # Look for aunt/uncle node
        IF node.parent == node.parent.parent.left
            aunt = node.parent.parent.right
            …
        ELSE
            aunt = node.parent.parent.left
            IF aunt.color == RED
                node.parent.color = BLACK
                aunt.color = BLACK
                node.parent.parent.color = RED
                node = node.parent.parent
```



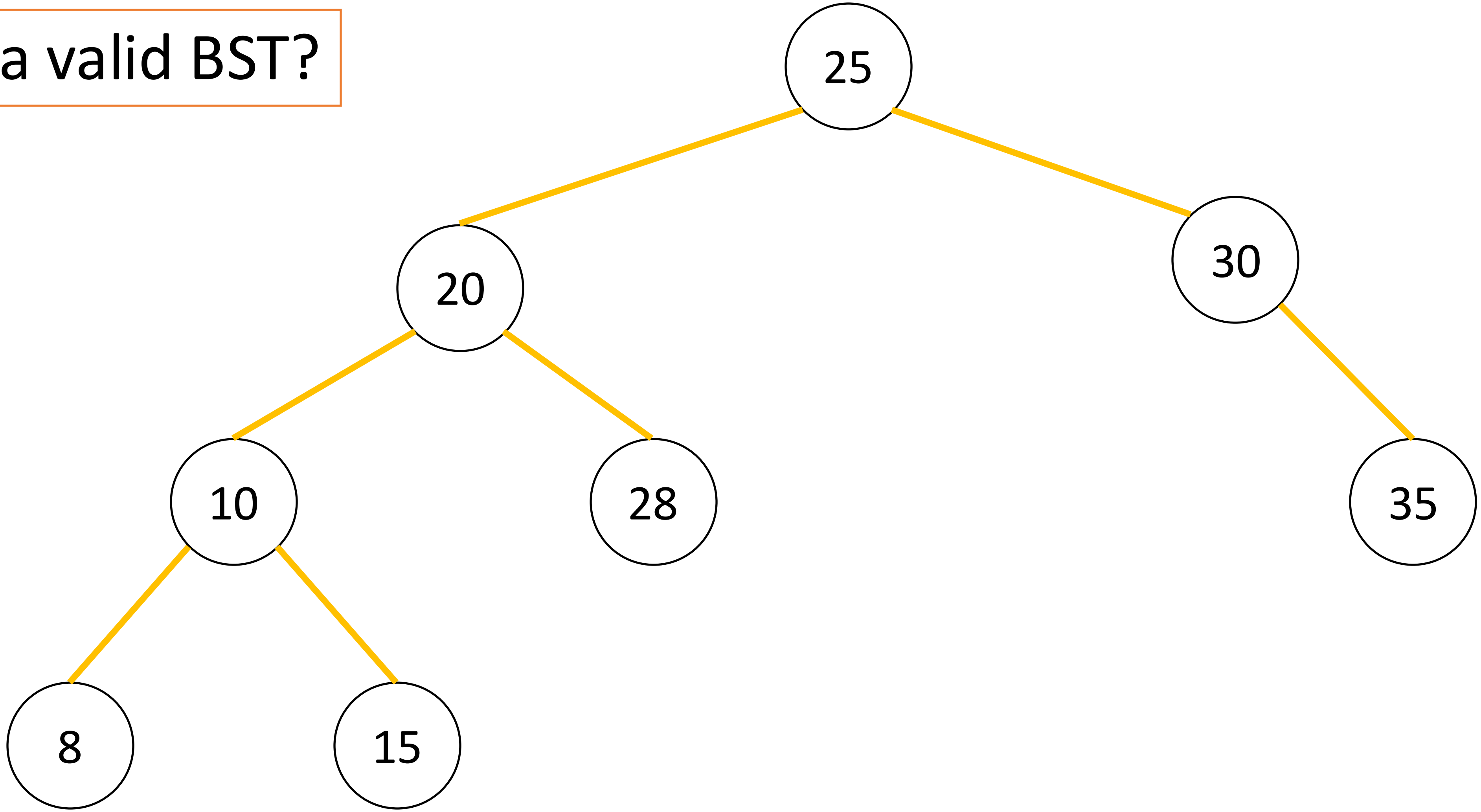
```
FUNCTION RBTreeFixColors(tree, node)
    WHILE node.parent.color == RED
        # Look for aunt/uncle node
        …
        ELSE
            aunt = node.parent.parent.left
            …
            ELSE
                IF node == node.parent.left
                    node = node.parent
                    RightRotate(tree, node)
              node.parent.color = BLACK
              node.parent.parent.color = RED
              LeftRotate(tree, node.parent.parent)
```



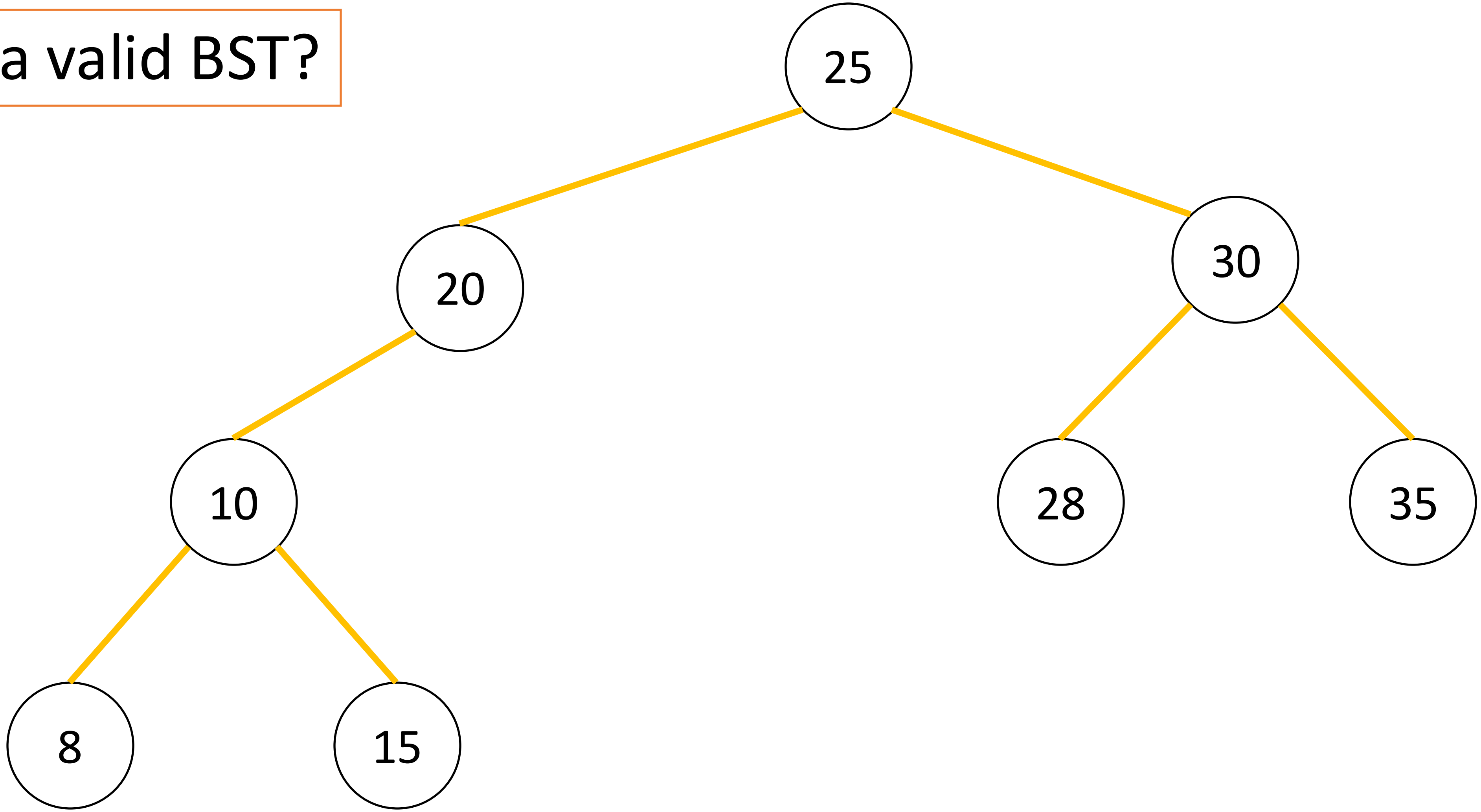
```
FUNCTION RBTreeFixColors(tree, node)
    WHILE node.parent.color == RED
        # Look for aunt/uncle node
        IF node.parent == node.parent.parent.left
            aunt = node.parent.parent.right
            …
        ELSE
            aunt = node.parent.parent.left
            …

    tree.root.color = BLACK
```



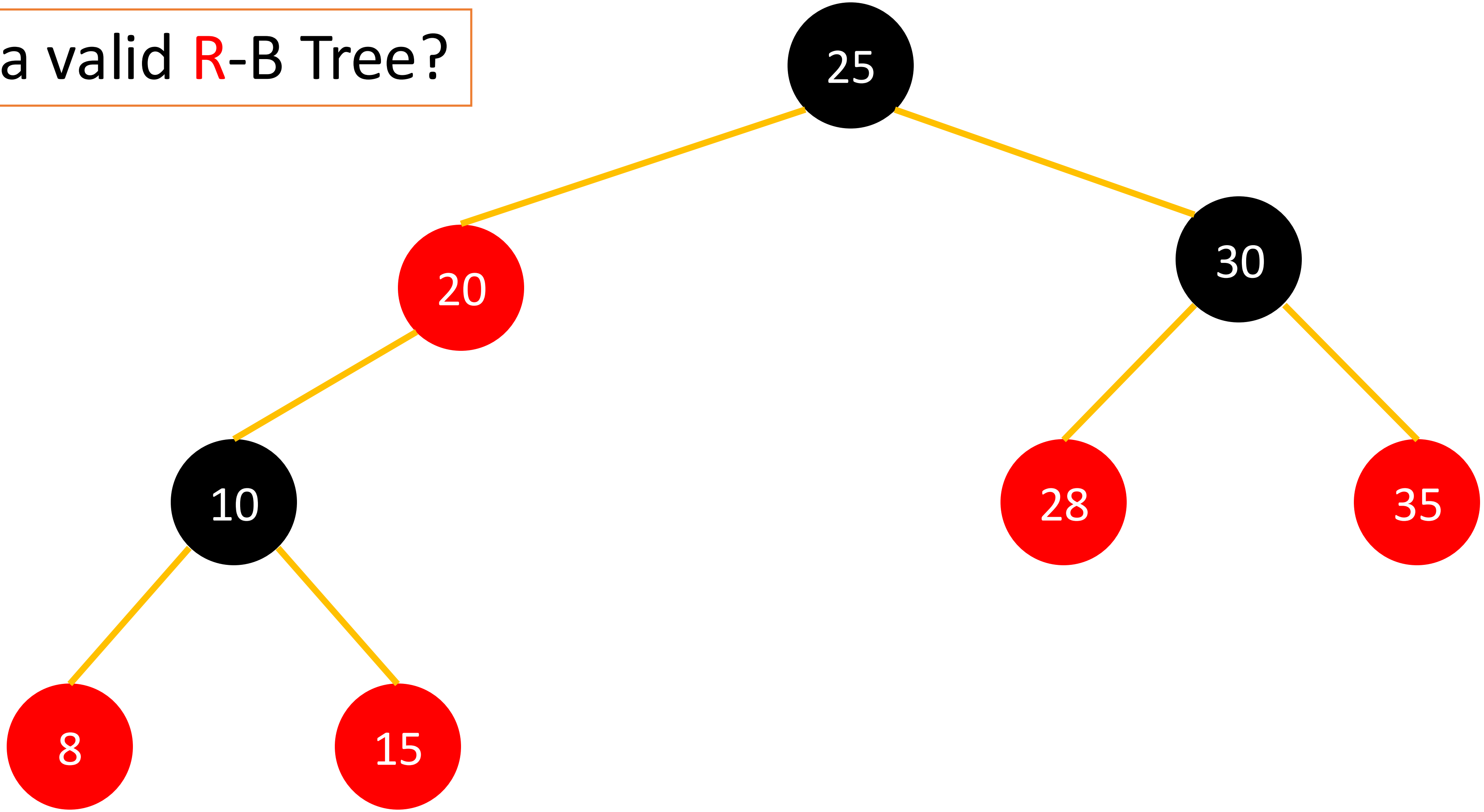
Is this a valid BST?



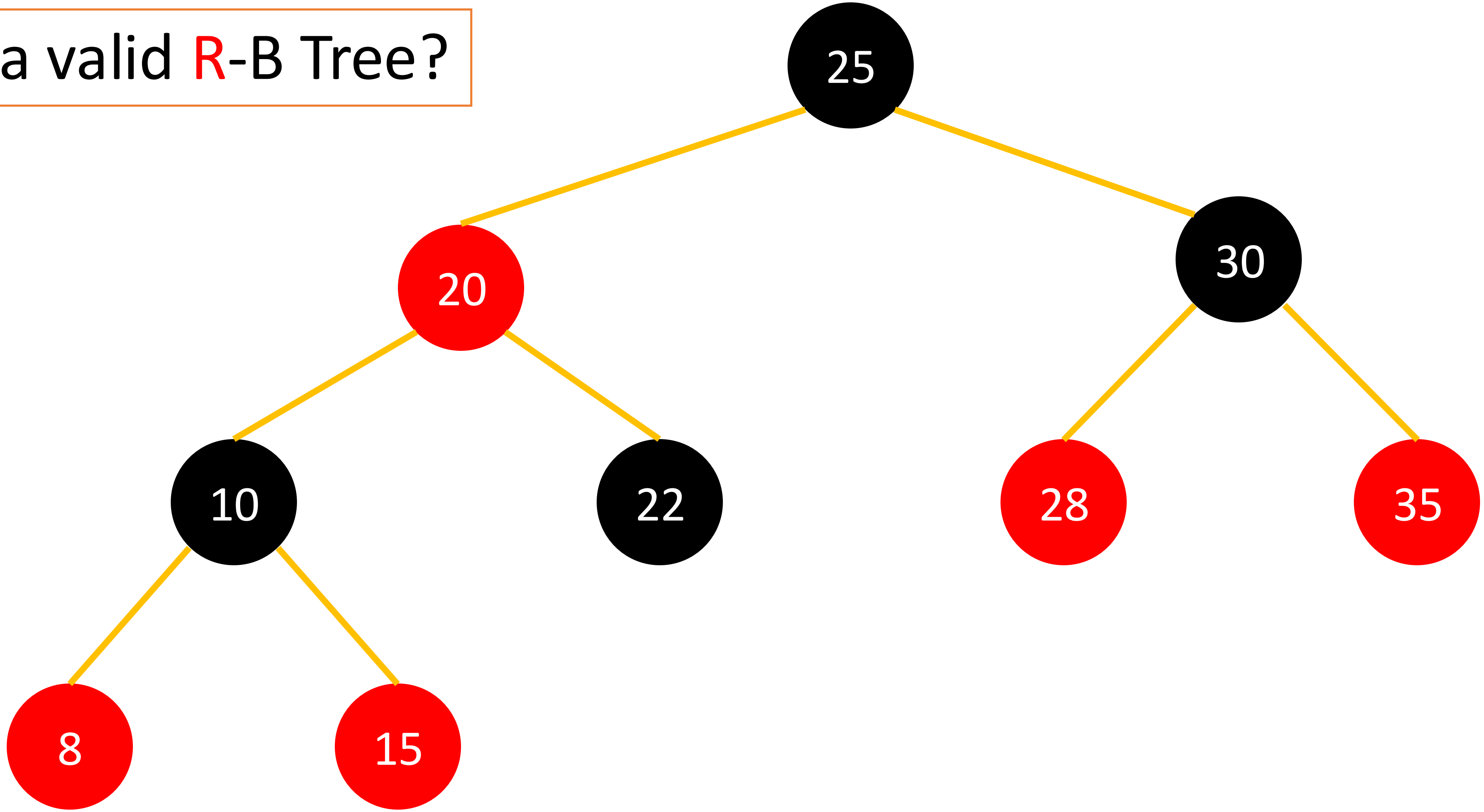
Is this a valid BST?



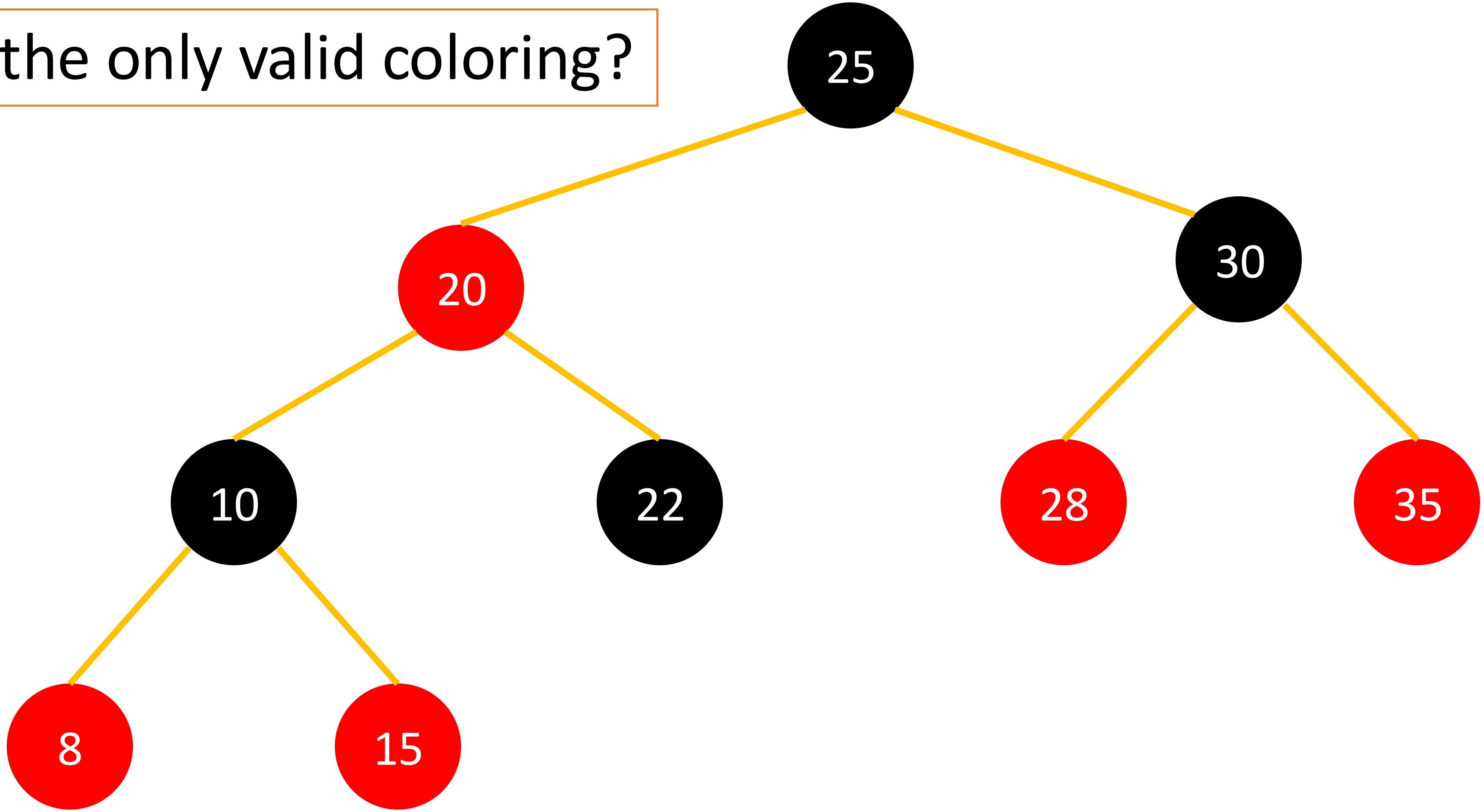
Is this a valid R-B Tree?



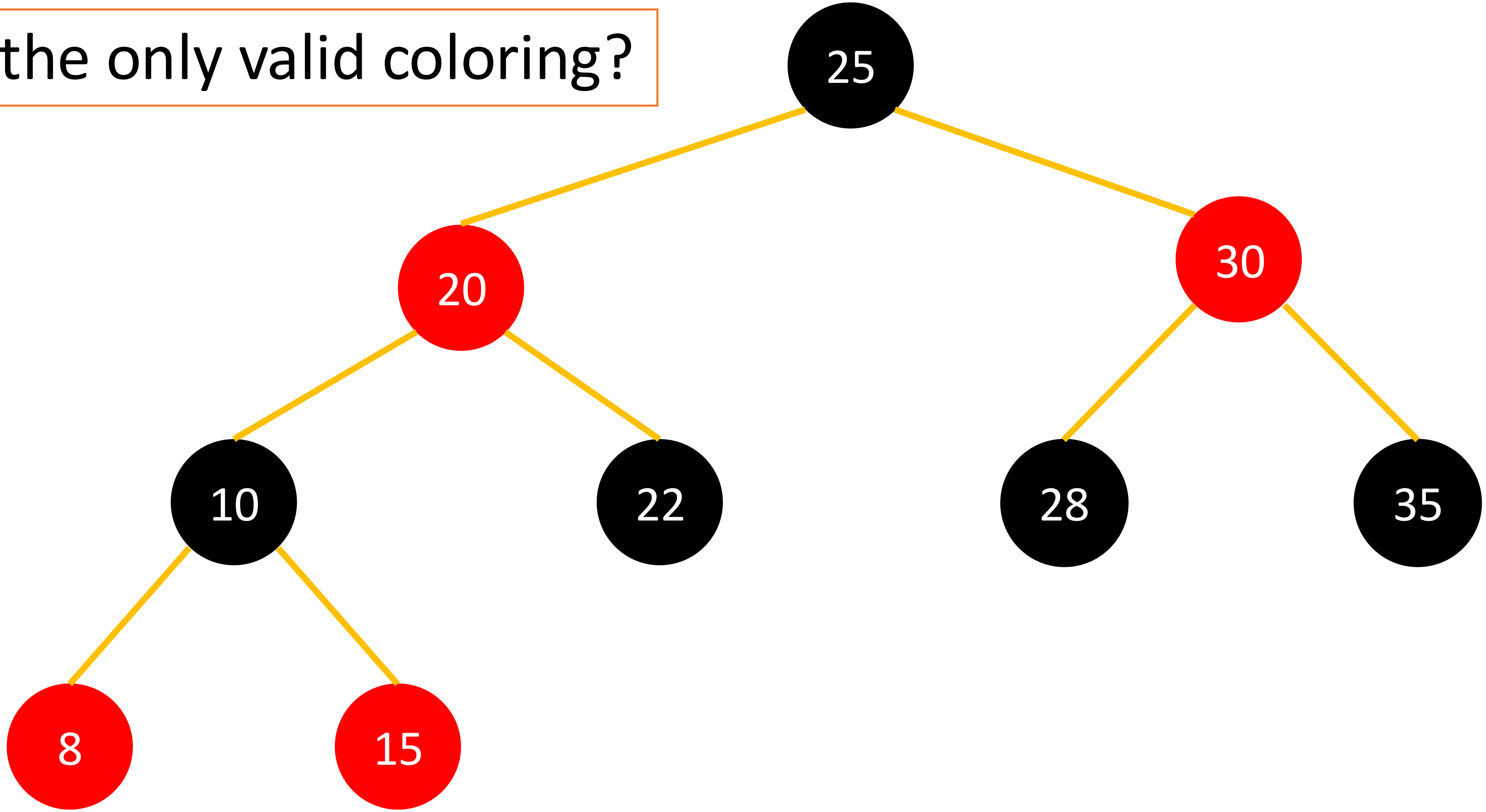
Is this a valid R-B Tree?



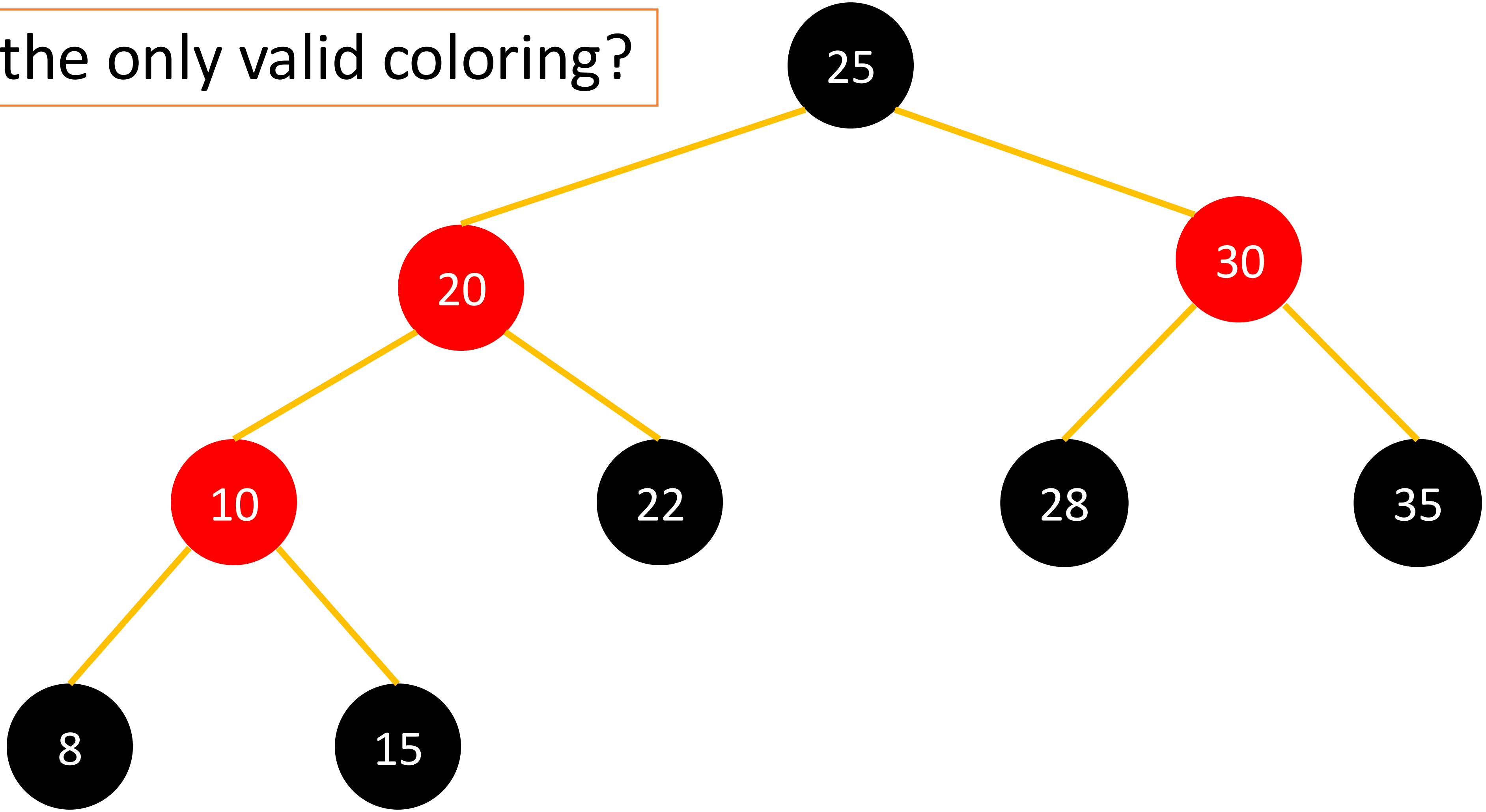
Is this the only valid coloring?



Is this the only valid coloring?

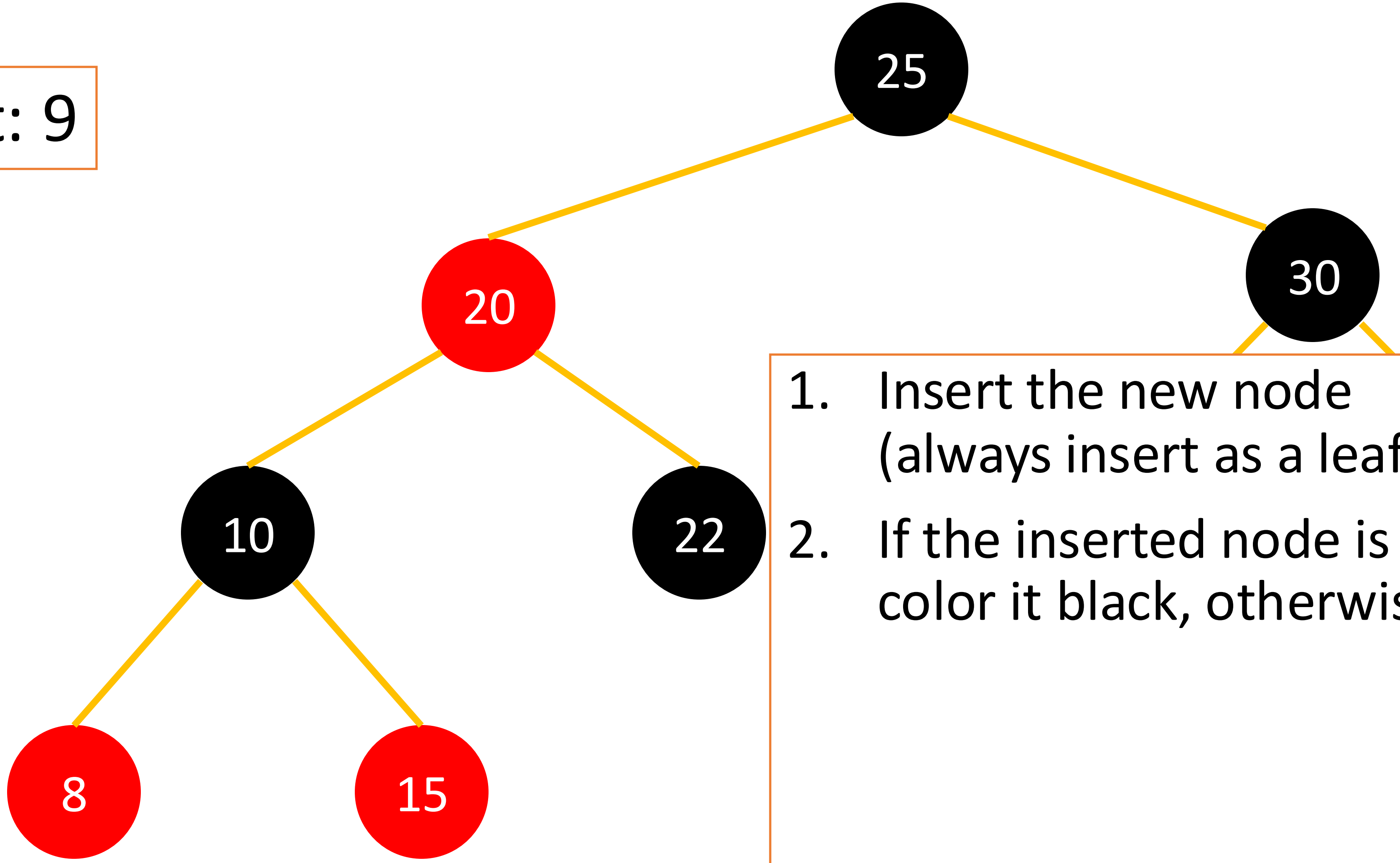


Is this the only valid coloring?



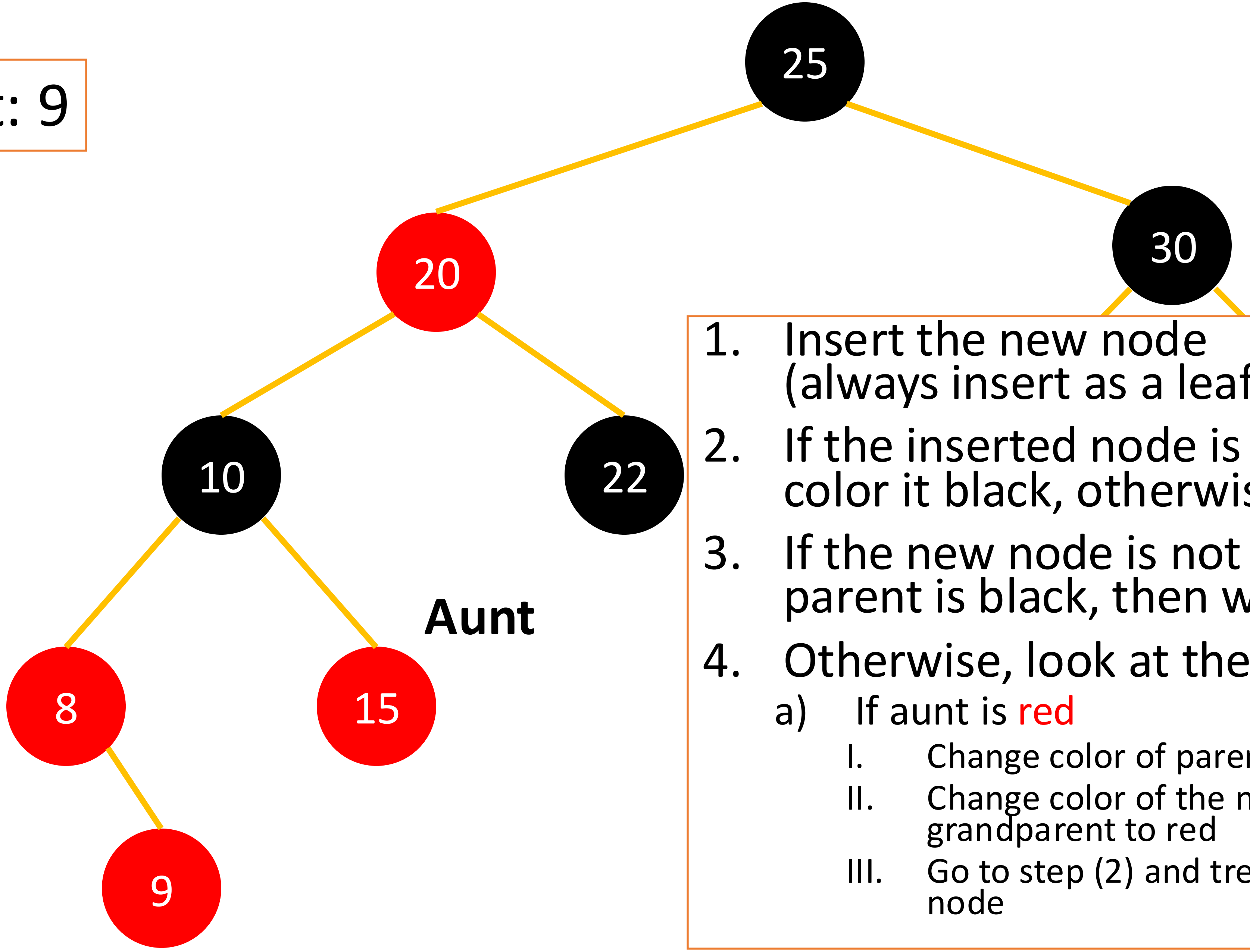
Exercise question 3

Insert: 9



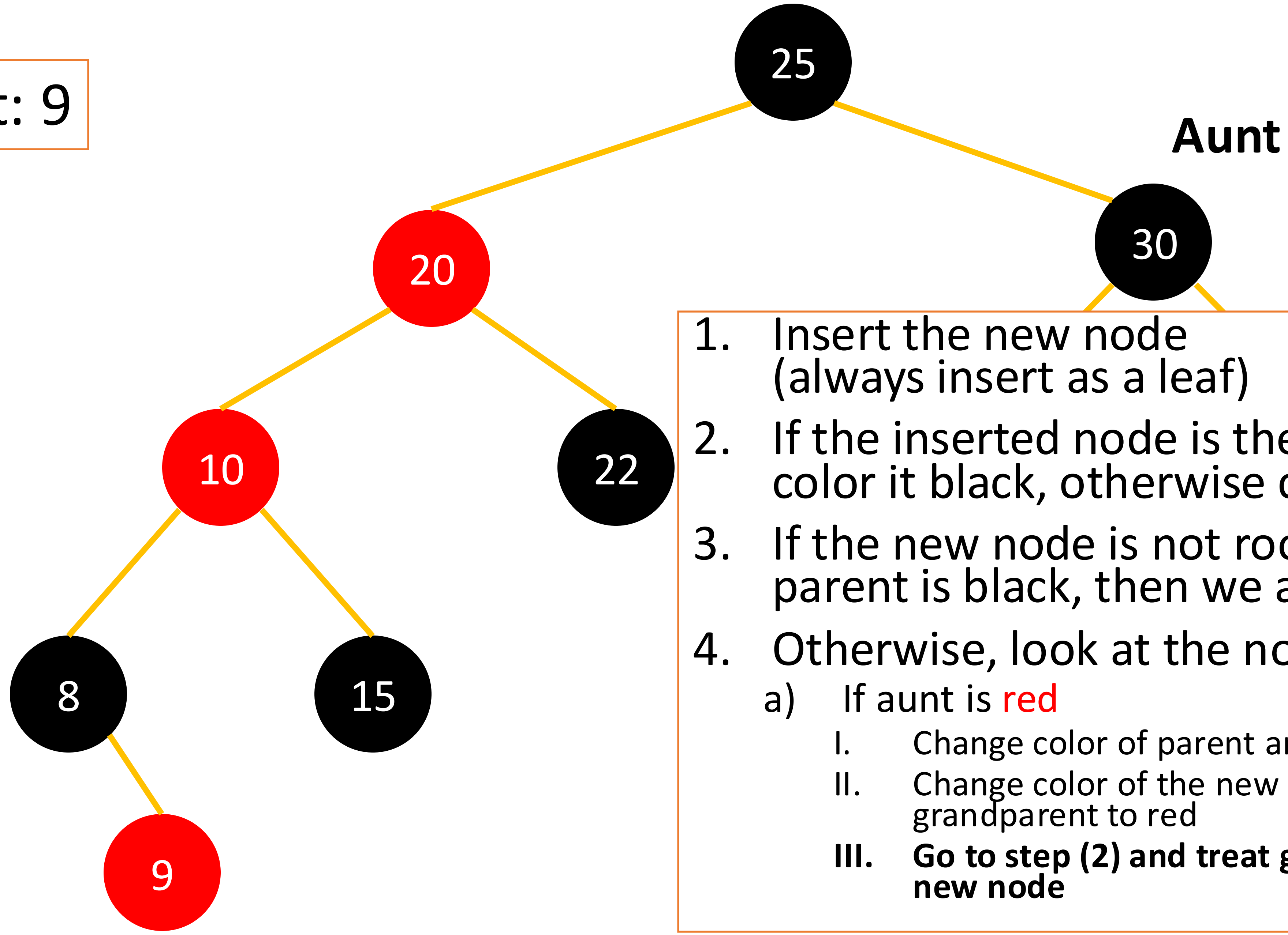
1. Insert the new node (always insert as a leaf)
2. If the inserted node is the root, then color it black, otherwise color it red

Insert: 9



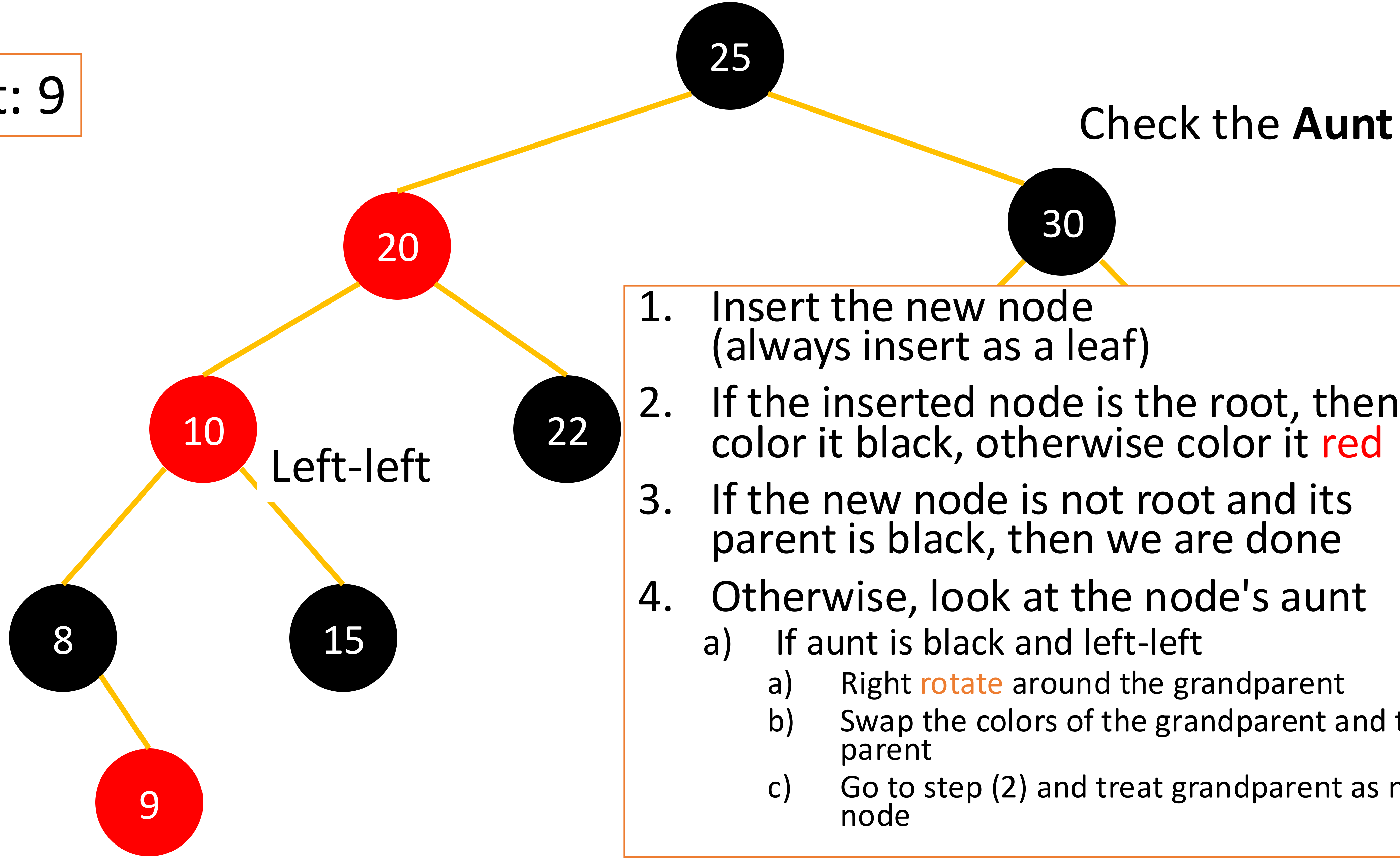
1. Insert the new node (always insert as a leaf)
2. If the inserted node is the root, then color it black, otherwise color it red
3. If the new node is not root and its parent is black, then we are done
4. Otherwise, look at the node's aunt
   a) If aunt is red
      I. Change color of parent and aunt to black
      II. Change color of the new node and the grandparent to red
      III. Go to step (2) and treat grandparent as new node

Insert: 9



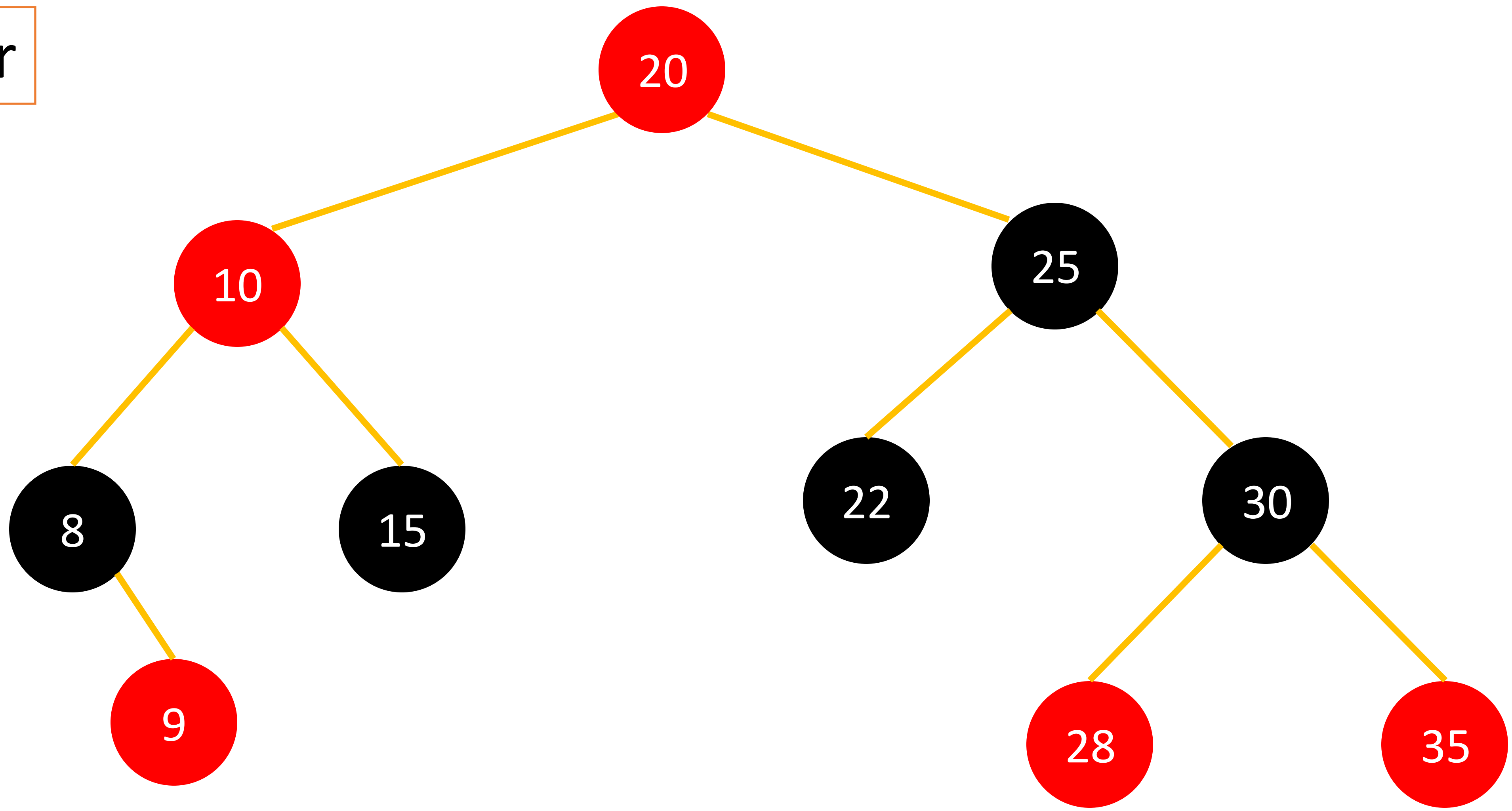
1. Insert the new node (always insert as a leaf)
2. If the inserted node is the root, then color it black, otherwise color it red
3. If the new node is not root and its parent is black, then we are done
4. Otherwise, look at the node's aunt
   a) If aunt is red
      I. Change color of parent and aunt to black
      II. Change color of the new node and the grandparent to red
      III. Go to step (2) and treat grandparent as new node

Insert: 9

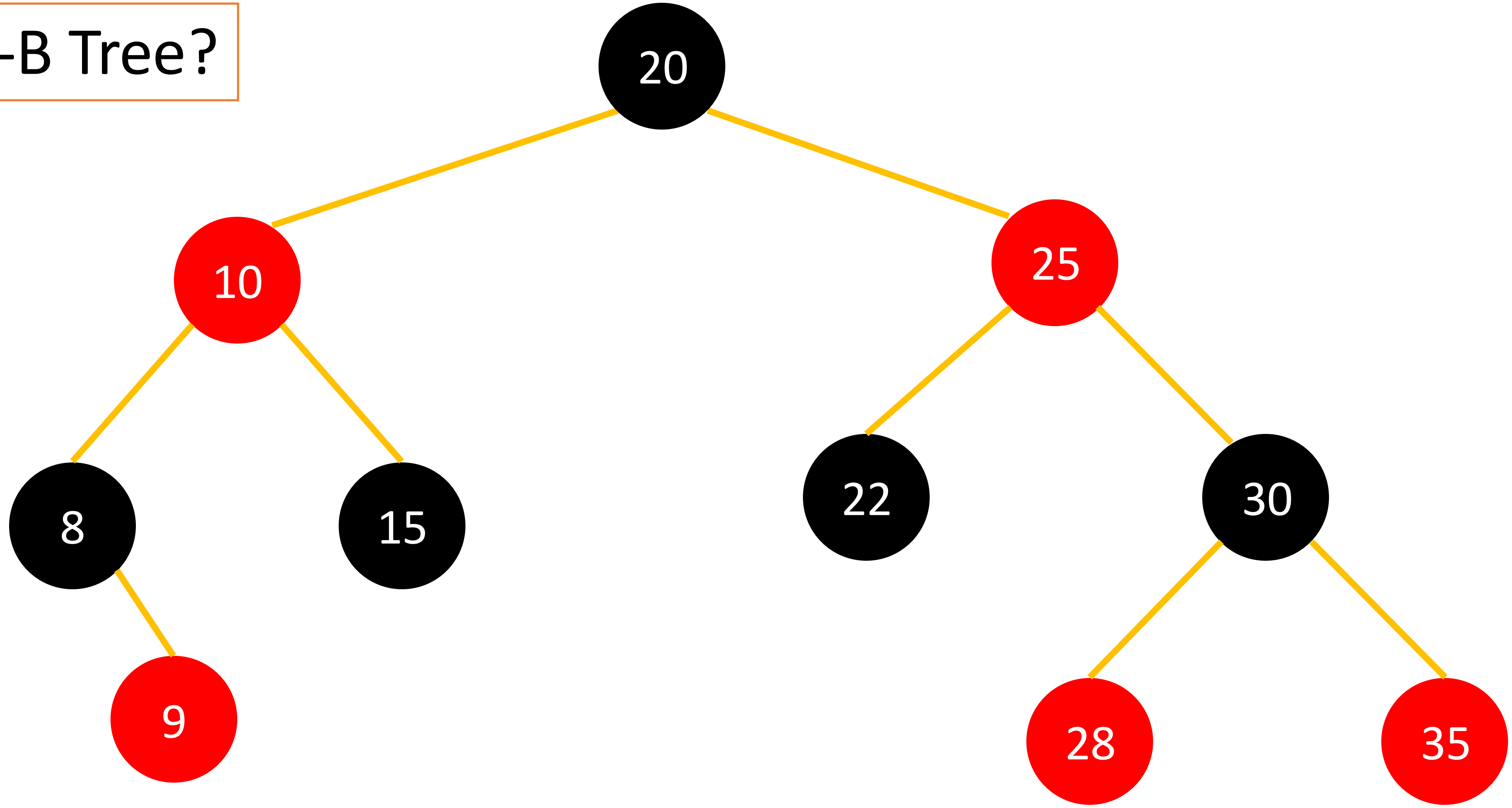


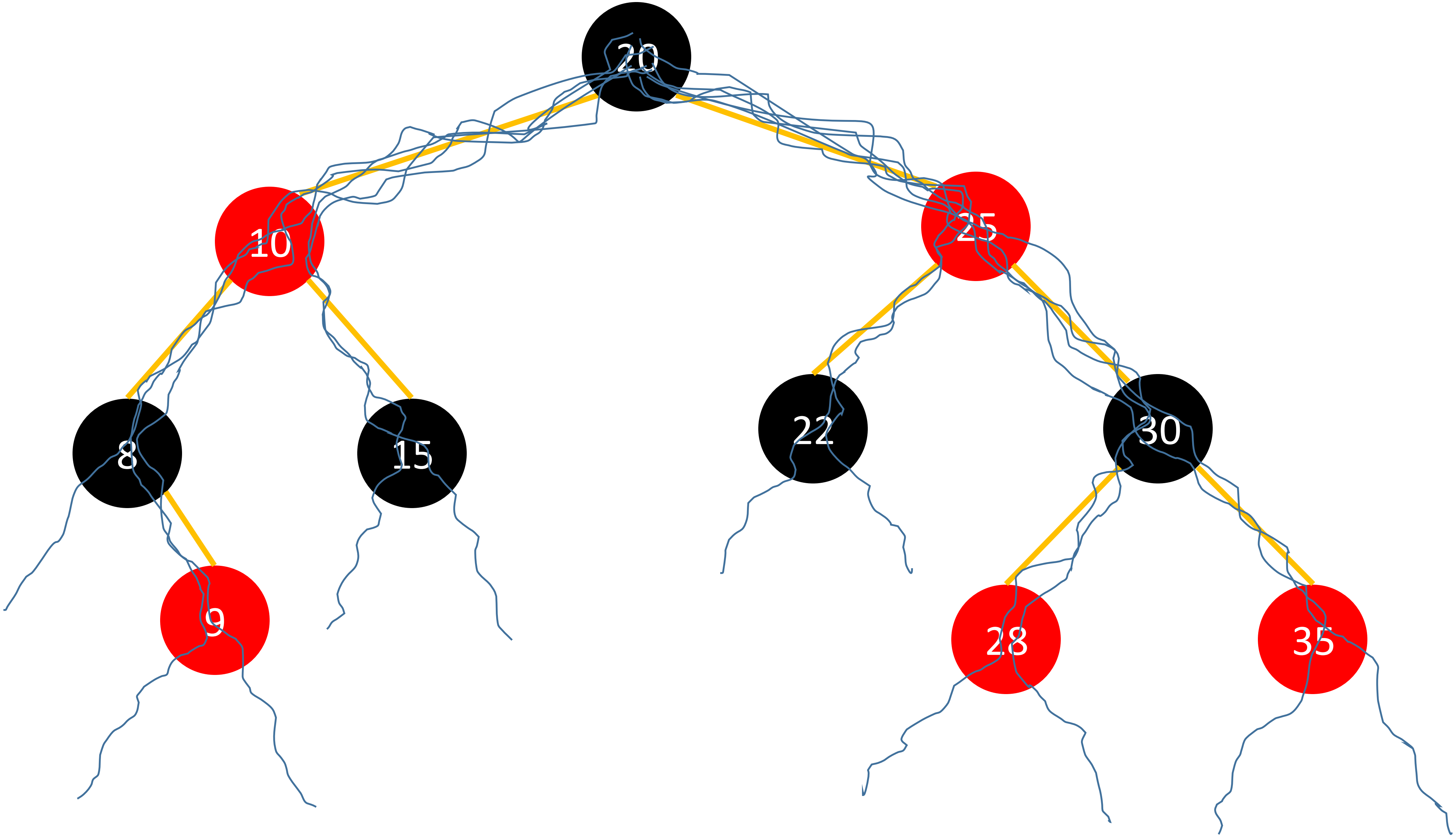
1. Insert the new node (always insert as a leaf)
2. If the inserted node is the root, then color it black, otherwise color it red
3. If the new node is not root and its parent is black, then we are done
4. Otherwise, look at the node's aunt
   a) If aunt is black and left-left
      a) Right rotate around the grandparent
      b) Swap the colors of the grandparent and the parent
      c) Go to step (2) and treat grandparent as new node

Recolor



Valid R-B Tree?




20
10
25
8
15
22
30
9
28
35

BST Summary

- Most BST operations take O(height) time.
- With an unbalanced tree this could be as bad as O(n)
- We want to ensure that the height of the tree is O(lg n)
- Red-Black trees provide one mechanism for creating balanced trees, meaning that they guarantee O(lg n) for applicable BST operation
- This requires extra work while inserting and deleting in the form of tree rotations

- Bottom line: as long as our tree satisfies the Red-Black tree invariants (which it does with appropriate insert/delete procedures), then we can assume optimal running time for BSTs