CSCI 232

Red Black Trees

Announcements

Lab 3 due **tonight**

Quiz 1 tomorrow. No lecture

No class on Monday (memorial day)

Quiz Logistics

Taken via D2L. You are not timed, but you have only one attempt
Opens 6:00 AM on Thursday, closes 11:59 PM on Thursday

10-15 Questions

- Short answer, multiple choice, true or false

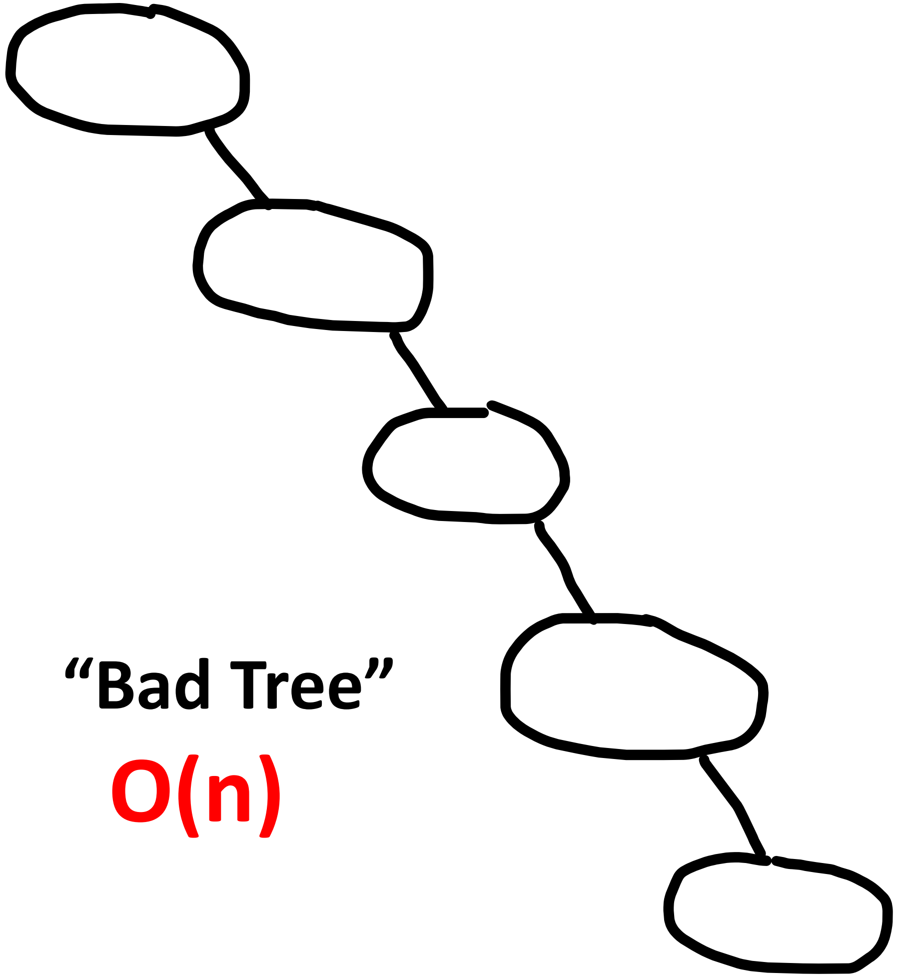
Quiz Content

- Basic Linked Lists, Arrays, Stacks and Queues
- General Trees
- Tree Traversal (Breadth-First, Depth-First, Inorder, postorder, preorder)
- BSTs and BST Functionality
- Applications of BSTs and Trees
- Time complexity of Tree/BST operations
- Red/Black Trees (purpose, how to verify if a tree is a R/B tree)

Lab 3 code

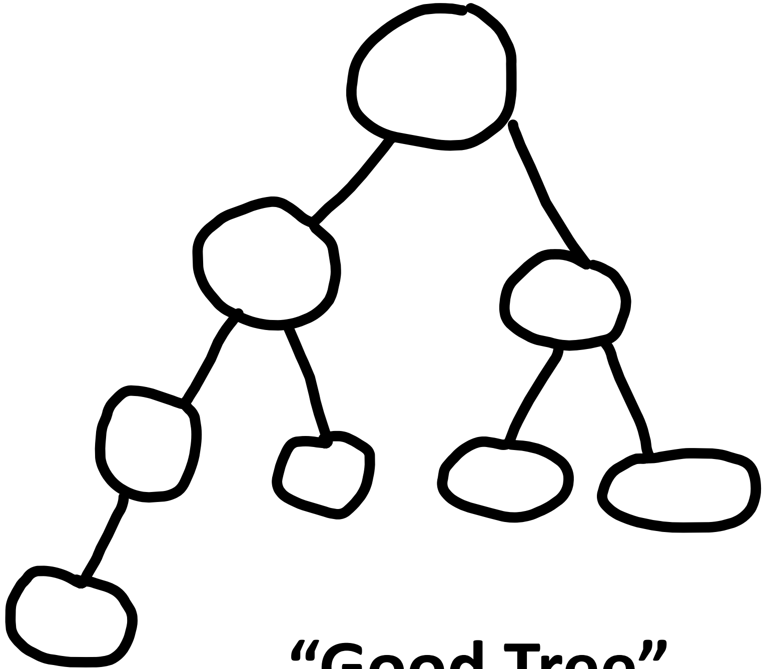
Binary Search Tree – Insertion/Searching/Removing

Running time?



“Bad Tree”

O(n)

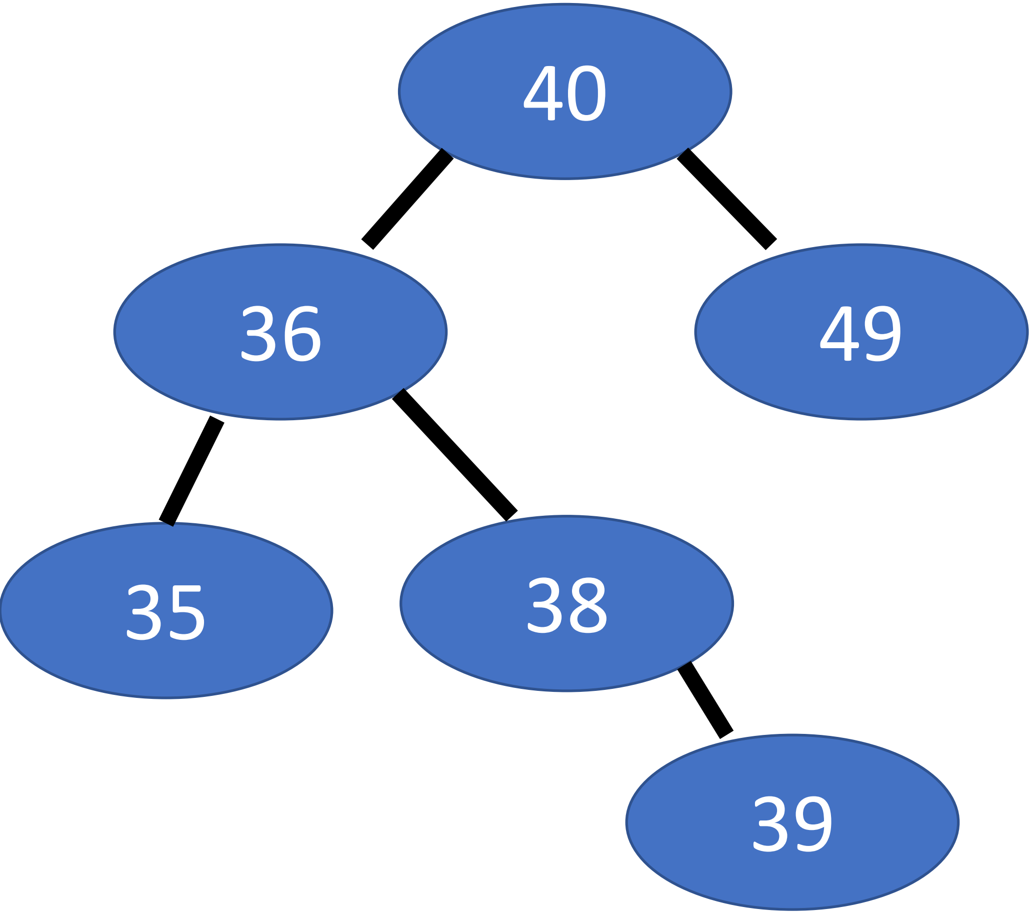


“Good Tree”

O(logn)

Balanced BST

A **balanced** binary tree, is defined as a binary tree in which given n nodes, the height of the tree is `O(logn)`.

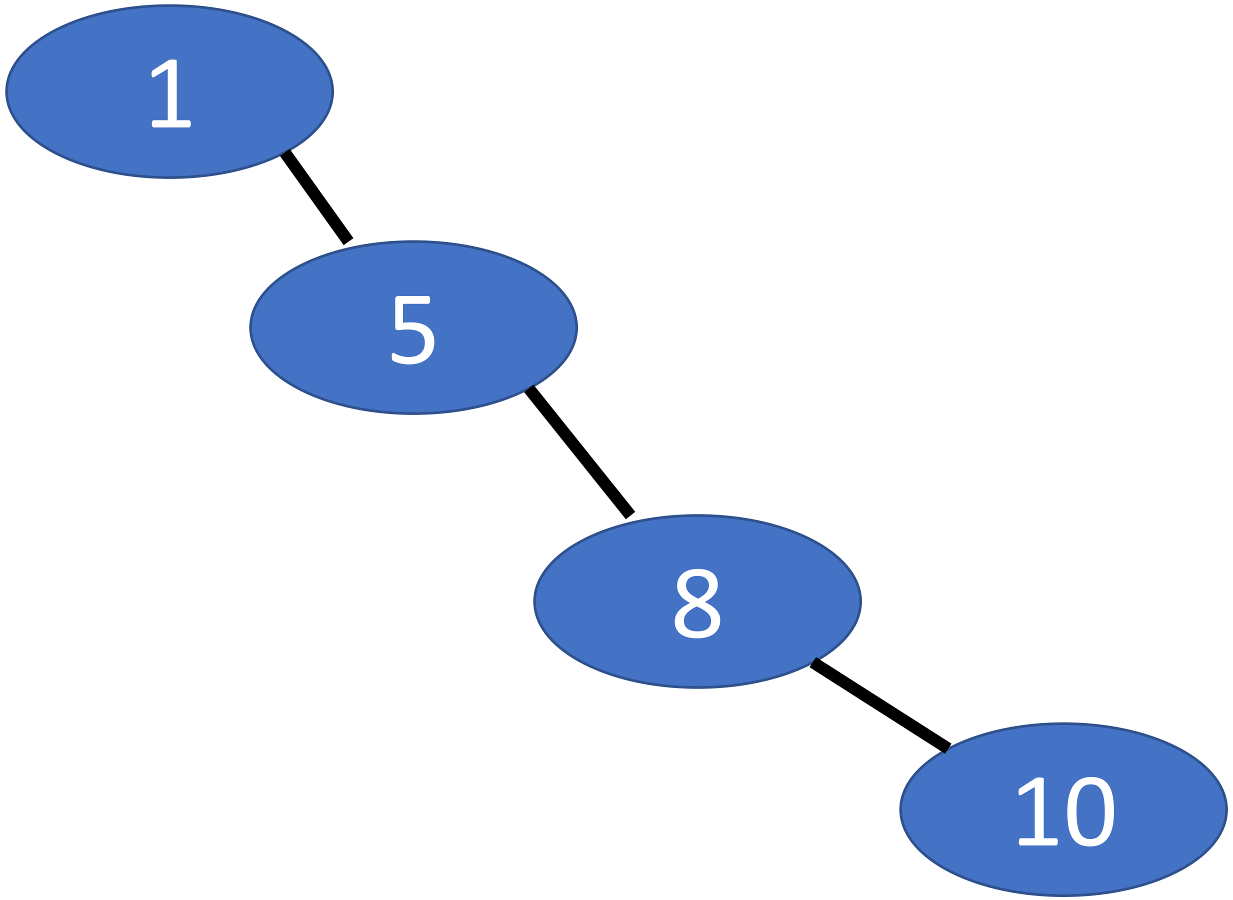


log(n)

Height	**Num Nodes**
0	1
1	2
2	4
3	8
4	16

Balanced BST

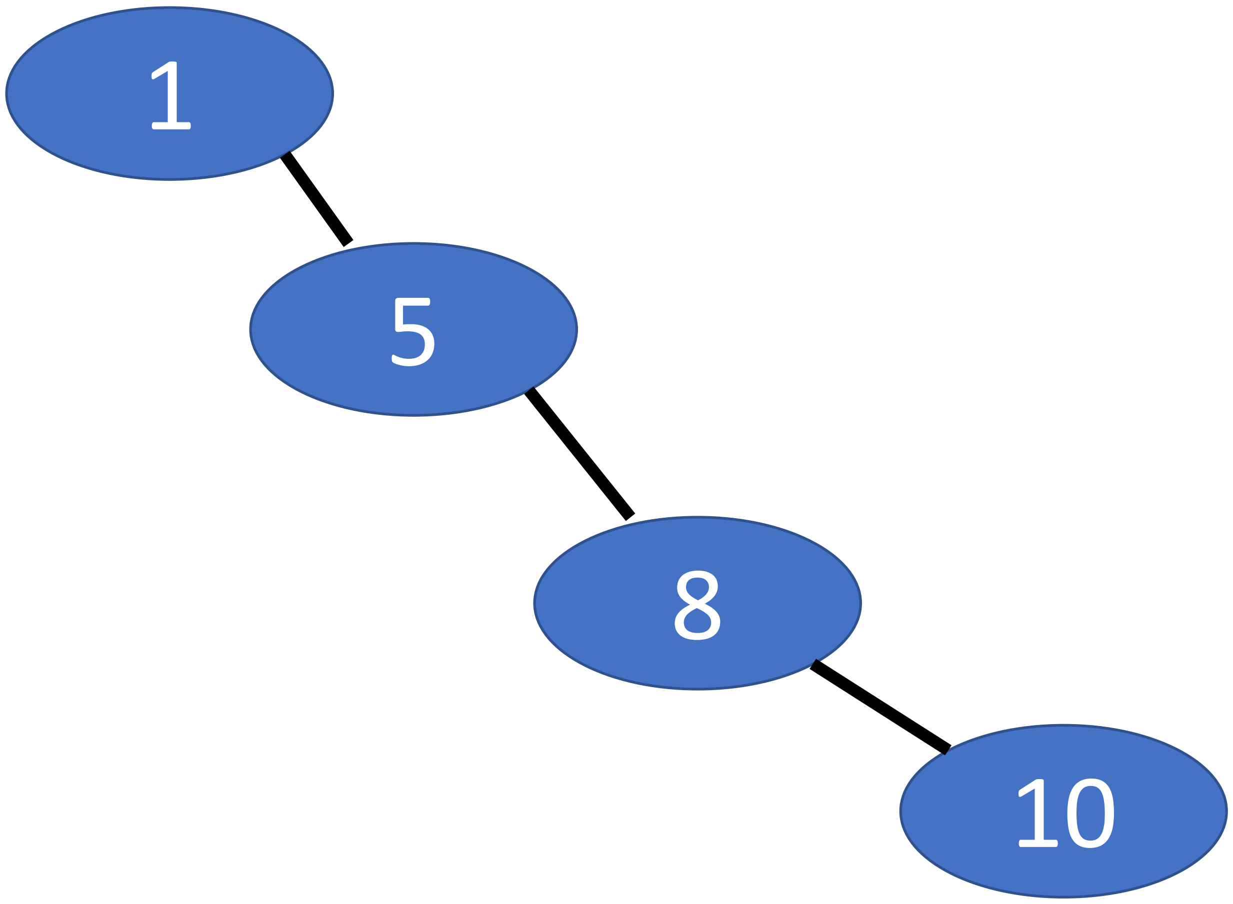
A **balanced** binary tree, is defined as a binary tree in which given n nodes, the height of the tree is **`O(logn)`**.



4 nodes
→ If this is a balanced tree, the height should be less than or equal to 2 (log(4))

Balanced BST

A **balanced** binary tree, is defined as a binary tree in which given n nodes, the height of the tree is `O(logn)`.

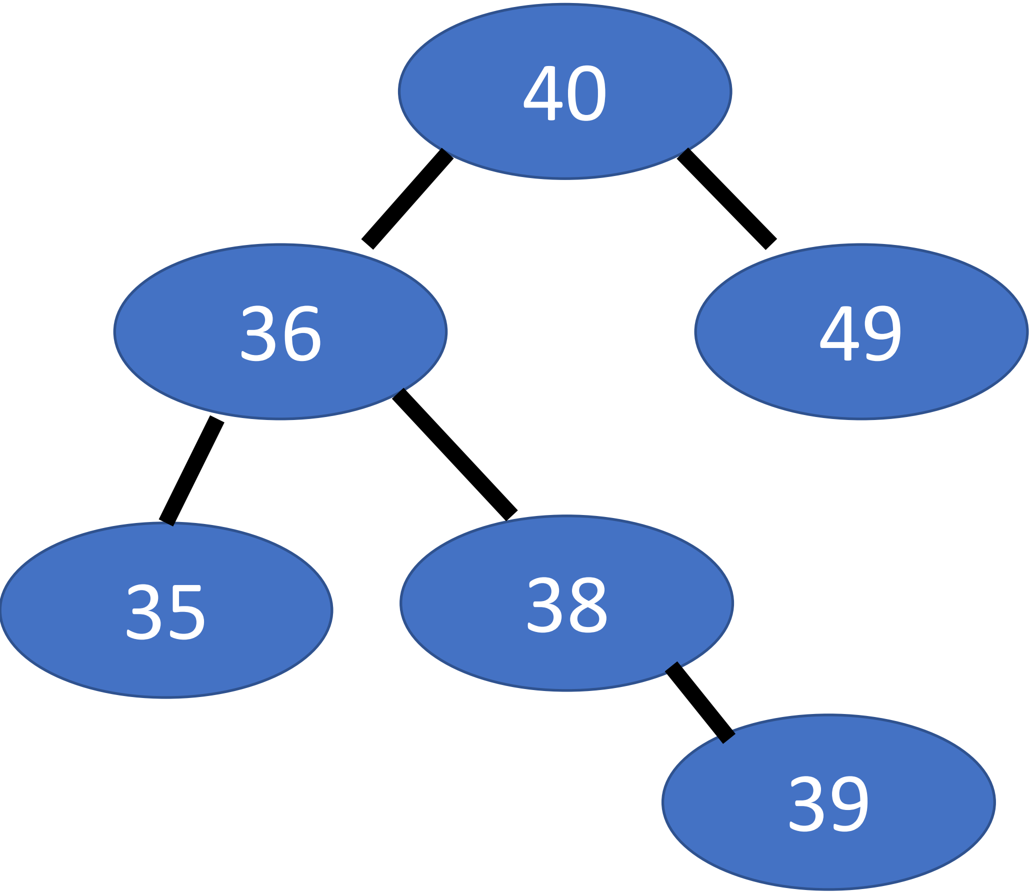


4 nodes
→ If this is a balanced tree, the height should be less than or equal to 2 (log(4))

Height = 3 → not balanced

Balanced BST

A **balanced** binary tree, is defined as a binary tree in which given n nodes, the height of the tree is **`O(logn)`**.

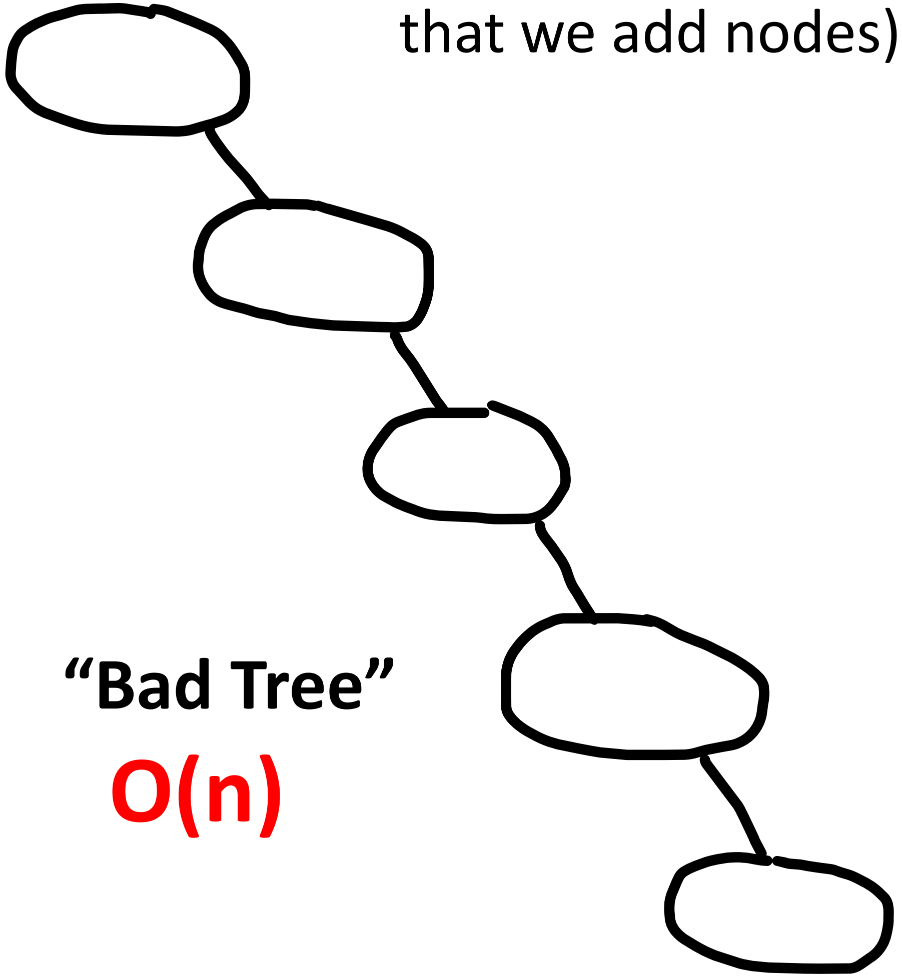


6 nodes
→ If this is a balanced tree, the height should be less than or equal to 3 ceil(log(6))

Height = 3 → balanced

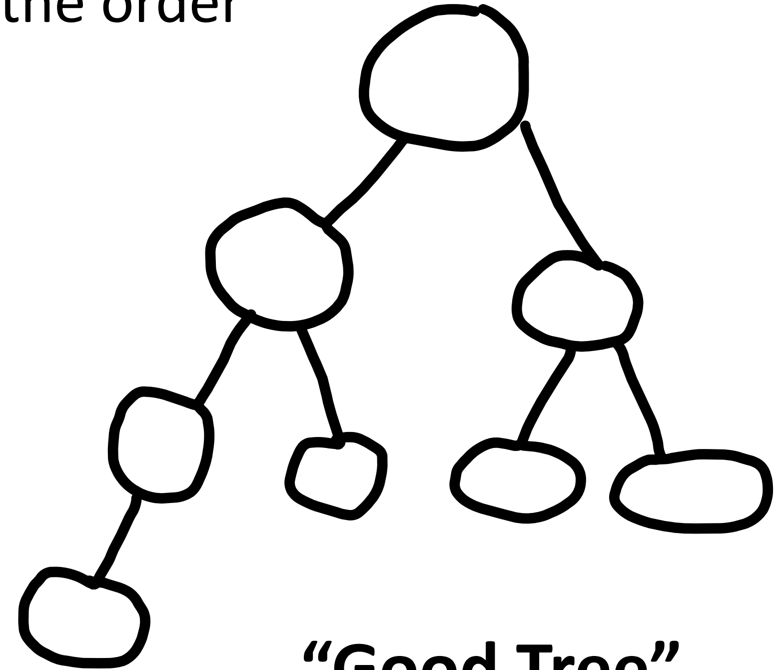
Balanced BST

If we are building a BST, there is no guarantee that the tree will be balanced (it depends on the order that we add nodes)



"Bad Tree"

O(n)



"Good Tree"

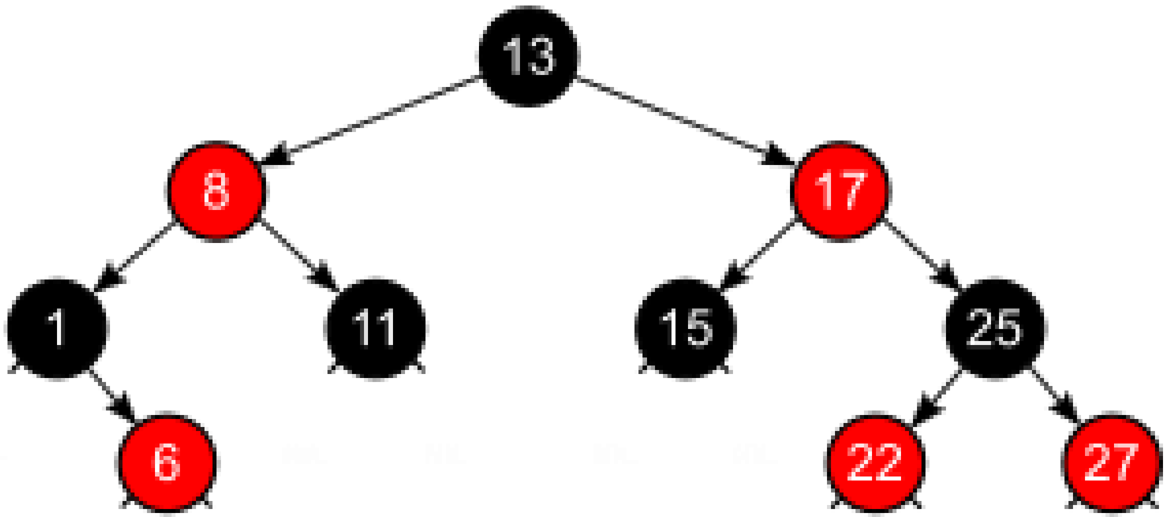
O(logn)

Balanced BST

Red-Black Trees are a type of BST with some more rules, and if we follow the rules, we will be guaranteed a balanced BST

Guaranteed Balanced BST =

- **O(logn)** insertion time
- **O(logn)** deletion time
- **O(logn)** searching time

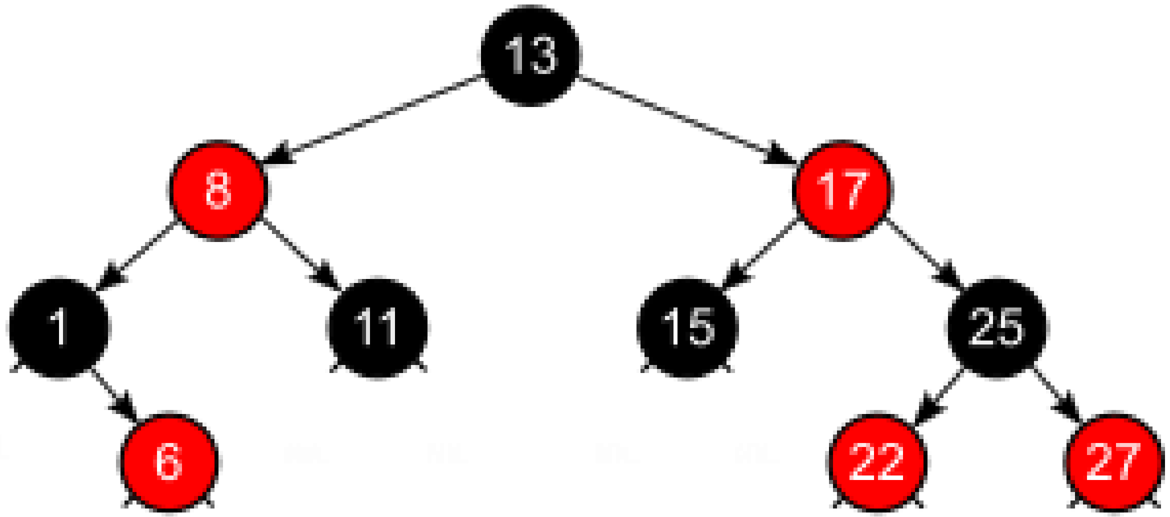


Red-Black Tree Rules

Because a RBT is a BST, we still need to make sure

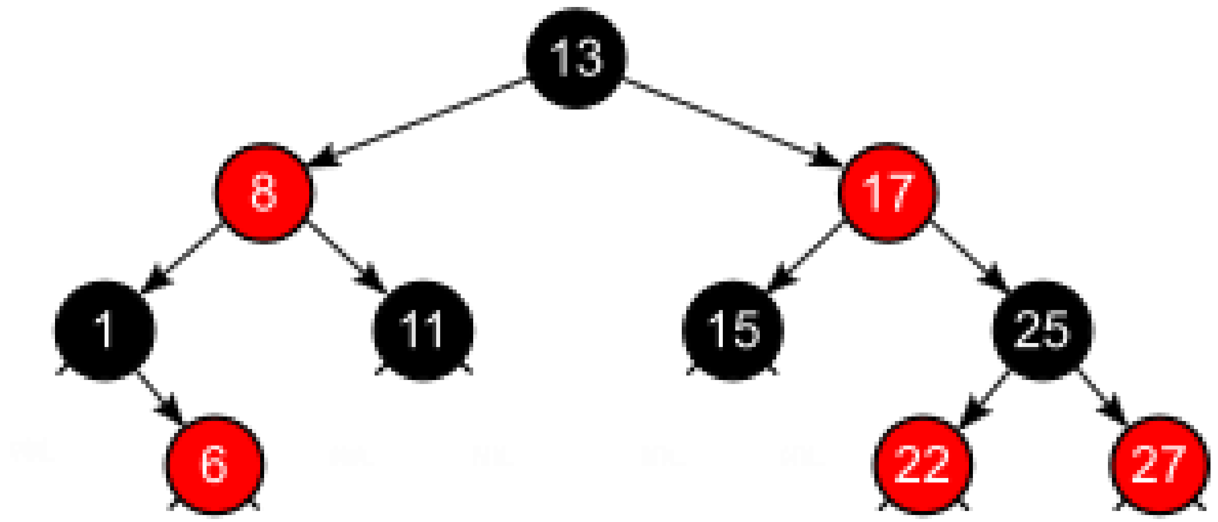
- Everything to the left of the node is less than the node
- Everything to the right of the node is less than the node
- A node cannot have more than two children
- No duplicate nodes

(BST Rules)



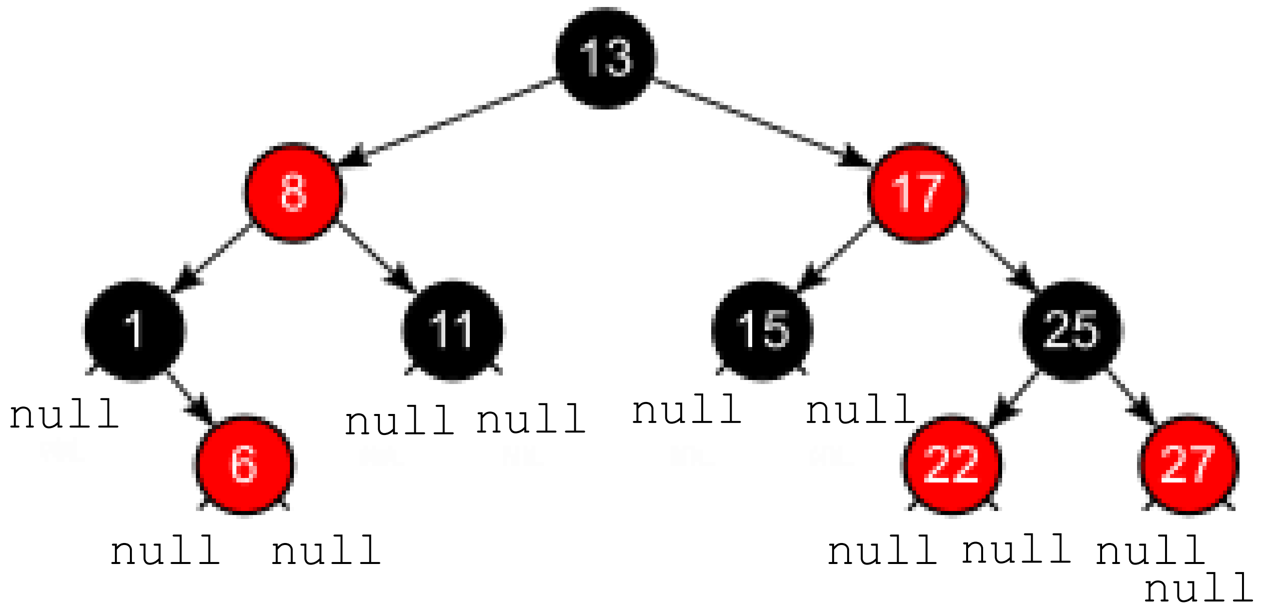
Red-Black Tree Rules

1. Every node is either red or **black**



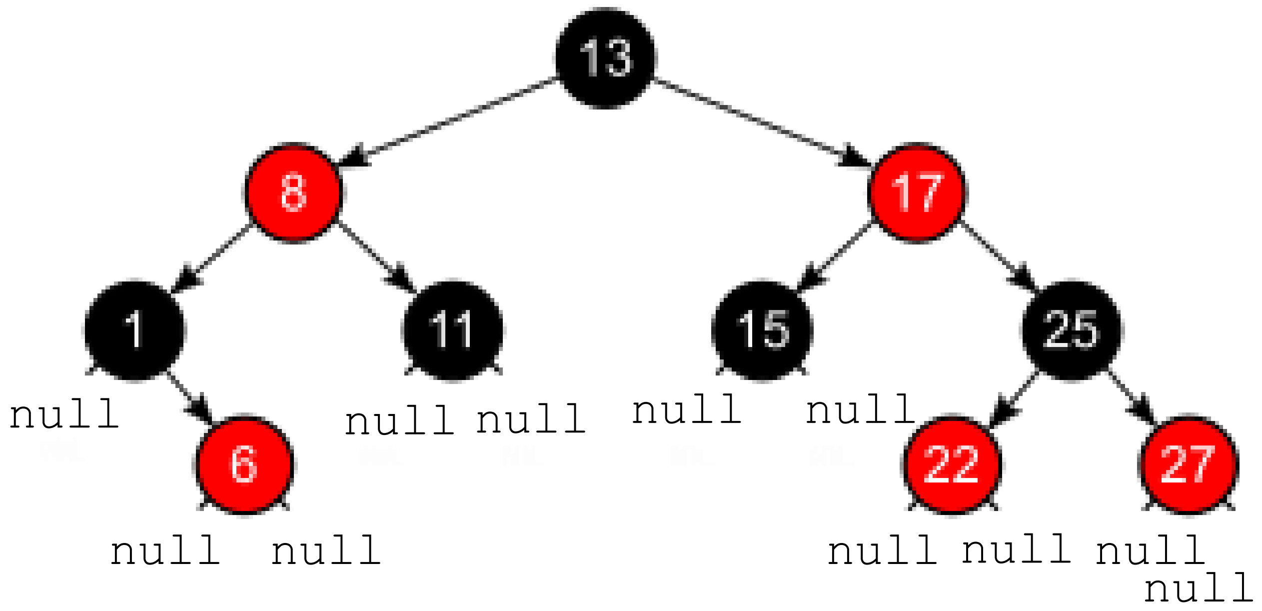
Red-Black Tree Rules

1. Every node is either **red** or **black**
2. The `null` children are **black**



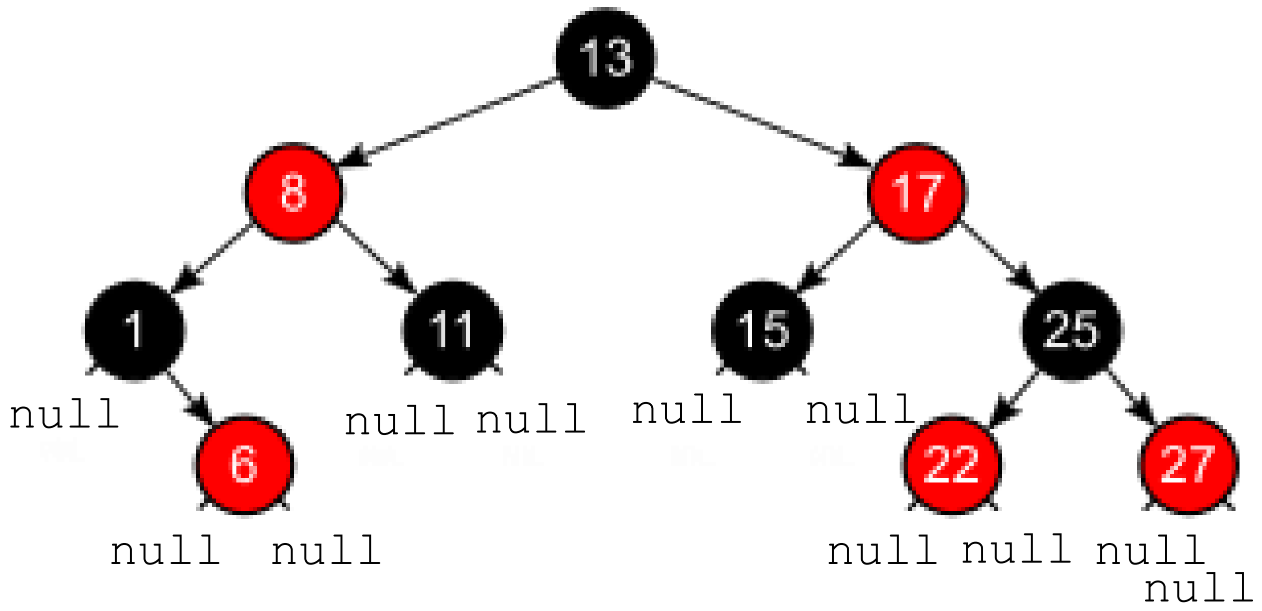
Red-Black Tree Rules

1. Every node is either red or **black**
2. The `null` children are **black**
3. The root node is **black**



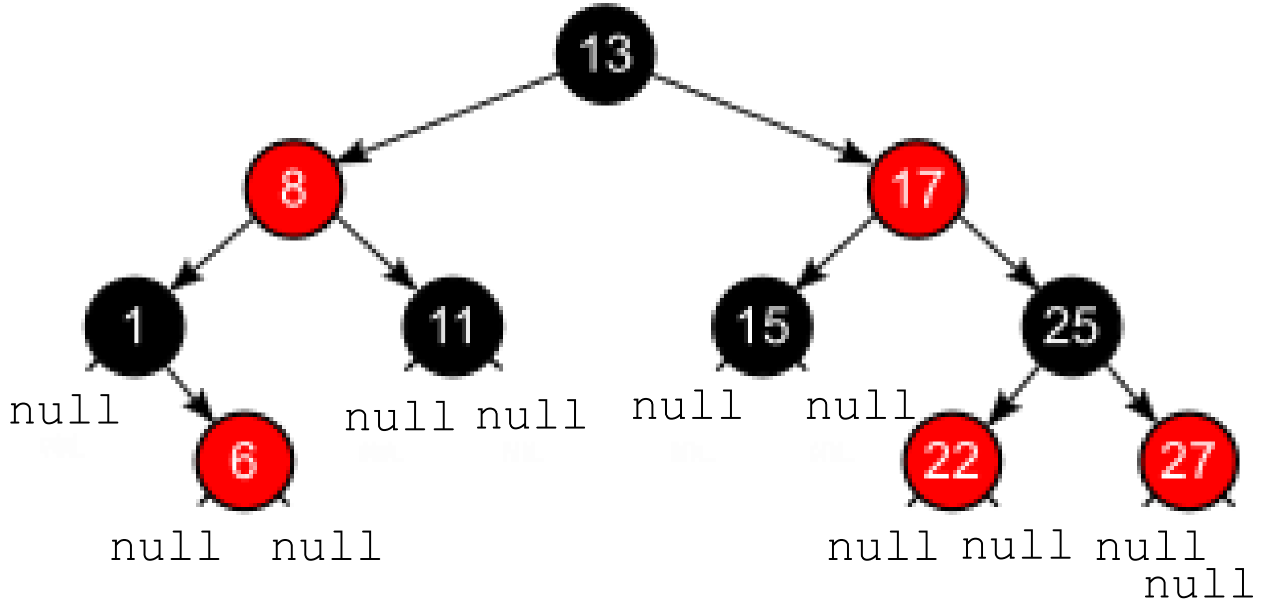
Red-Black Tree Rules

1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**
4. If a node is **red**, both children must be **black**



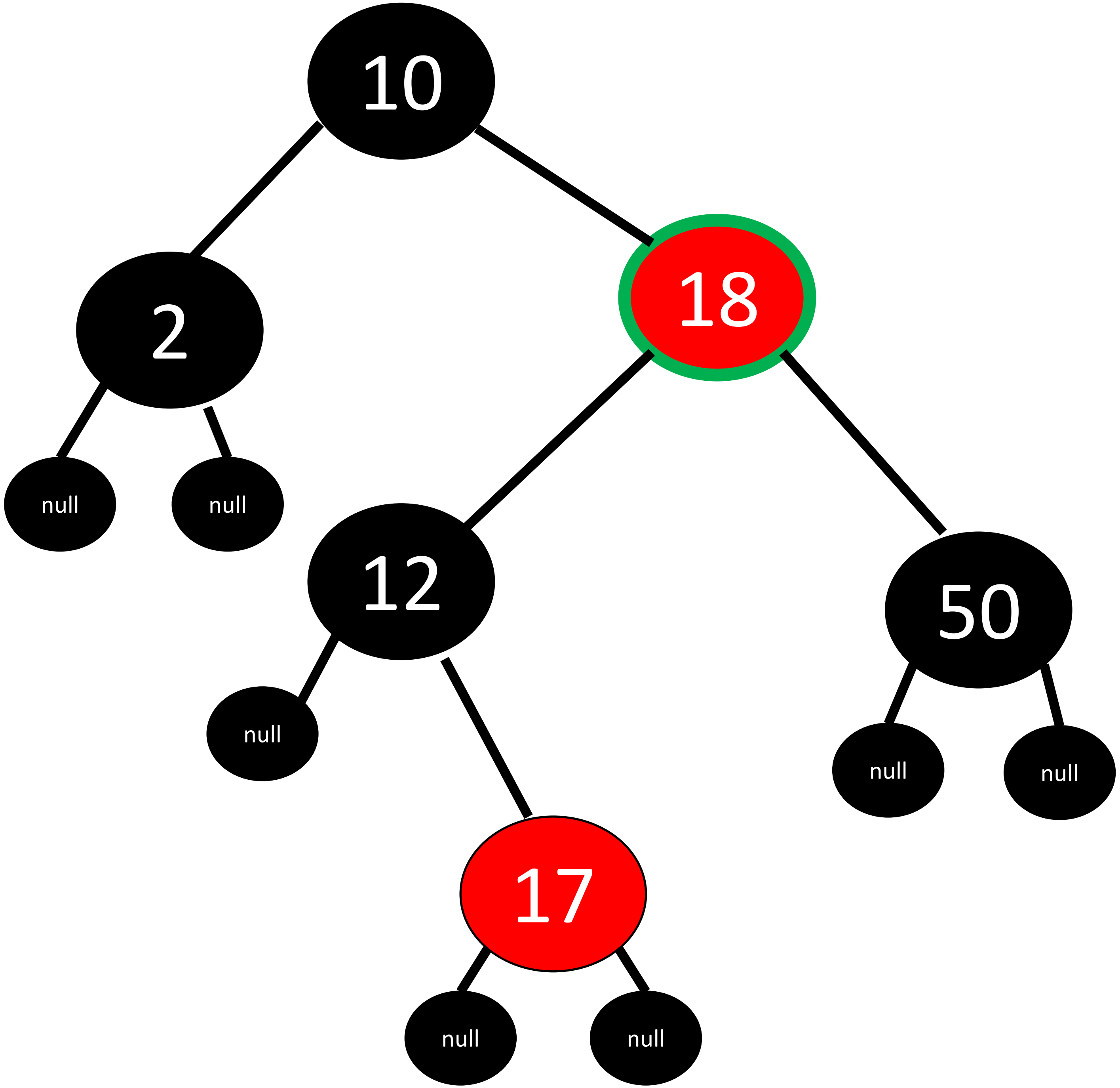
Red-Black Tree Rules

1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**
4. If a node is **red**, both children must be **black**
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes



Red-Black Tree Rules

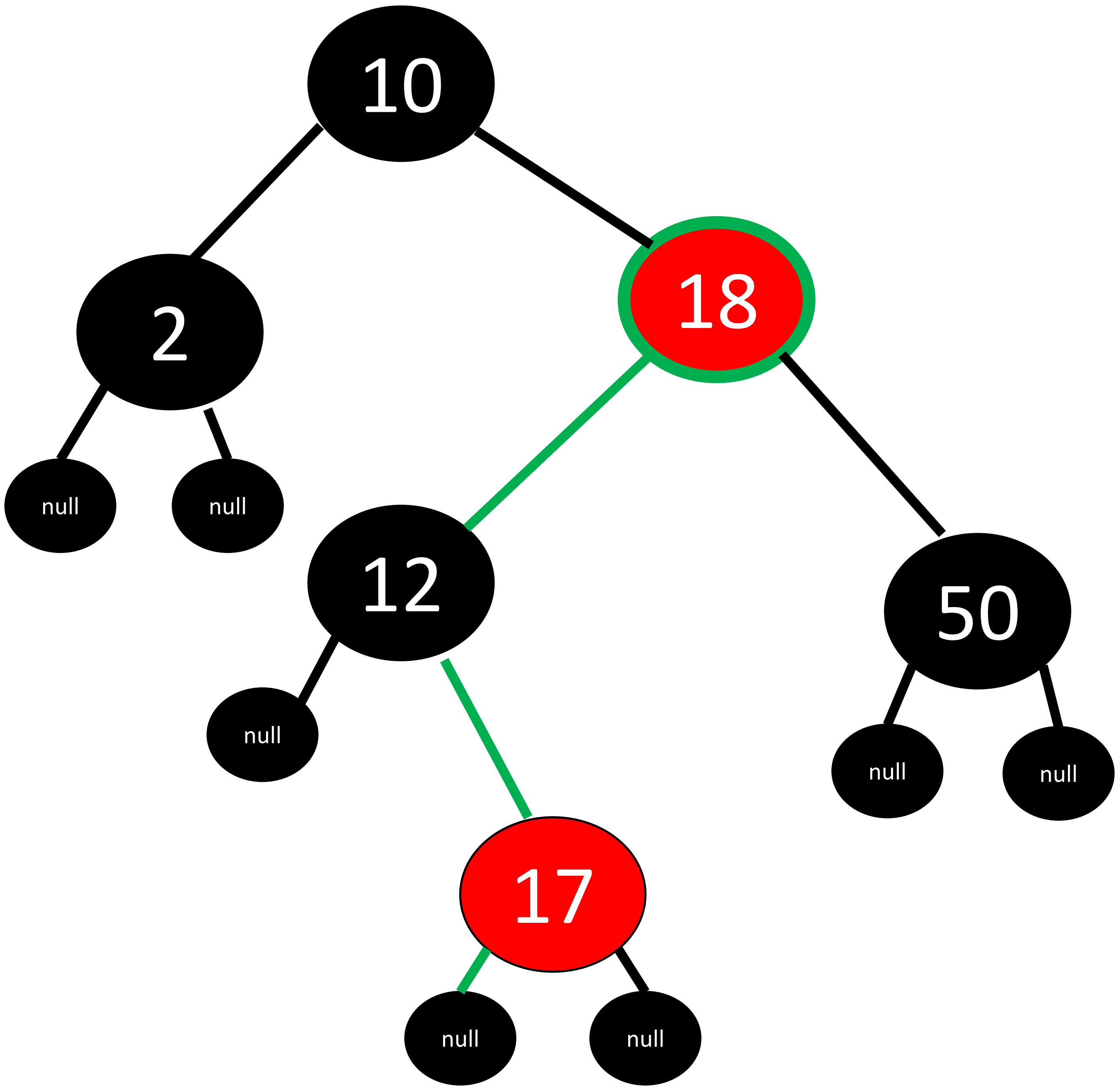
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes



Red-Black Tree Rules

5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

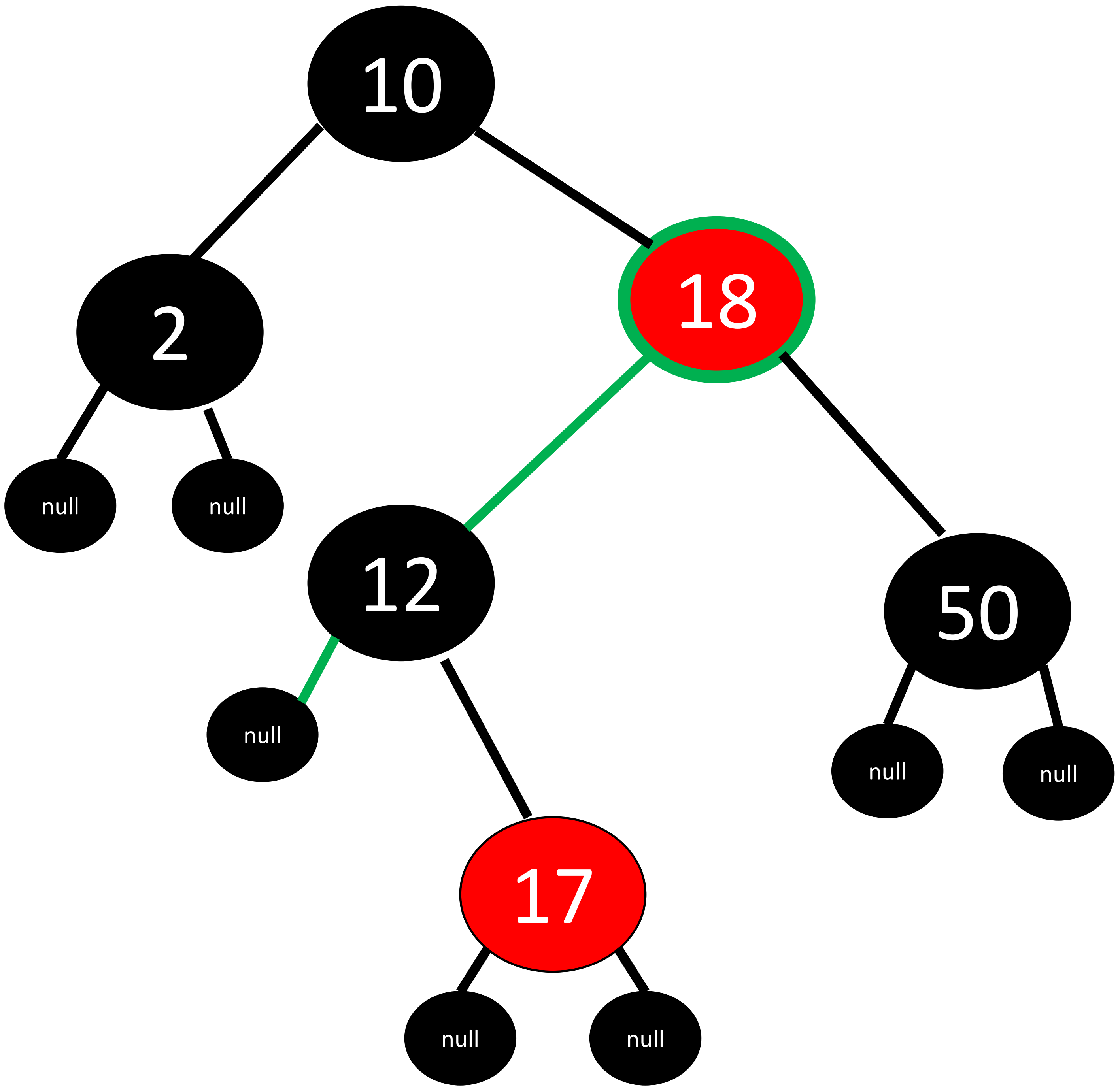
Path 1: 2 black nodes visited



Red-Black Tree Rules

5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

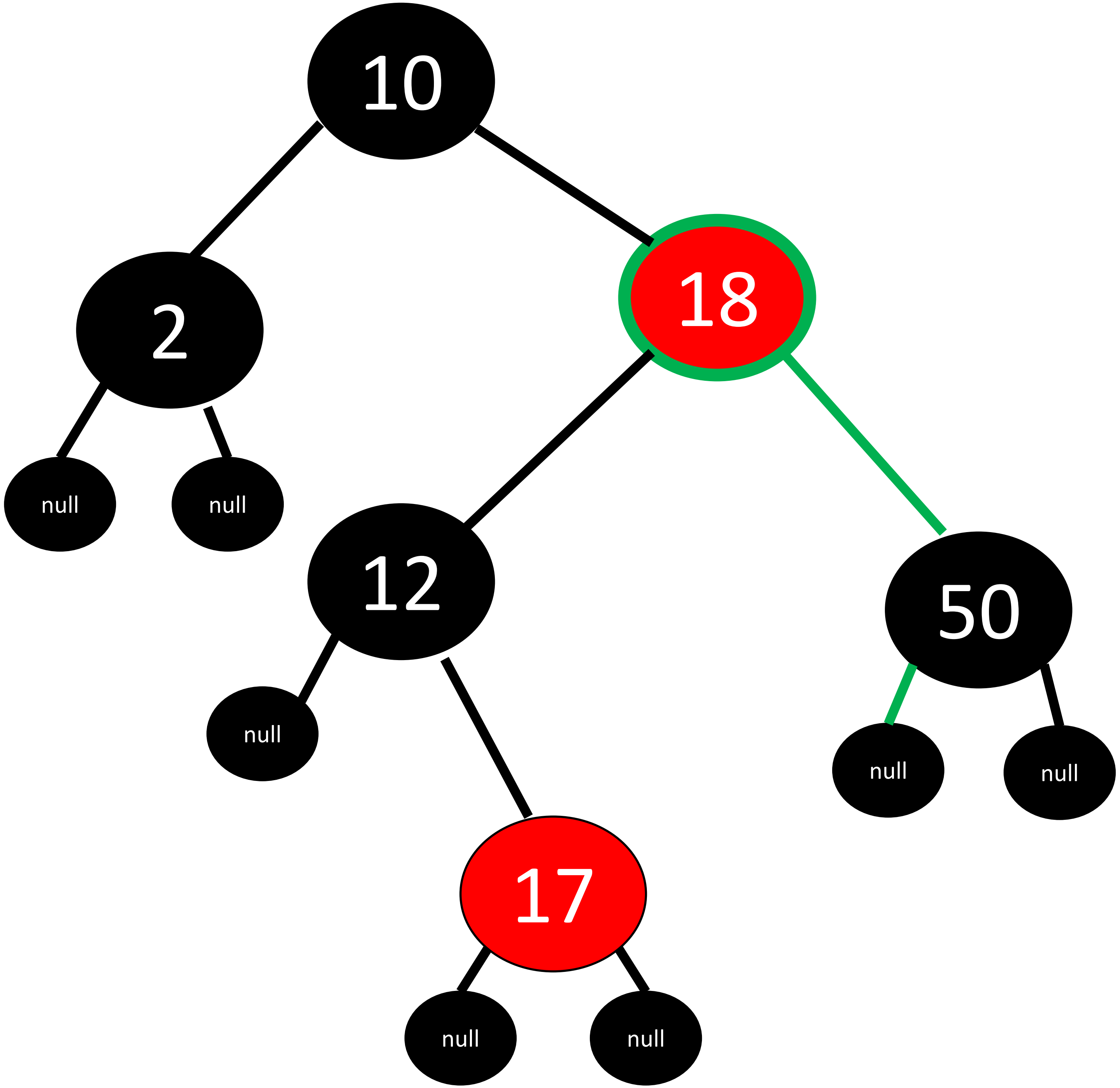
Path 1: 2 black nodes visited
Path 2: 2 black nodes visited



Red-Black Tree Rules

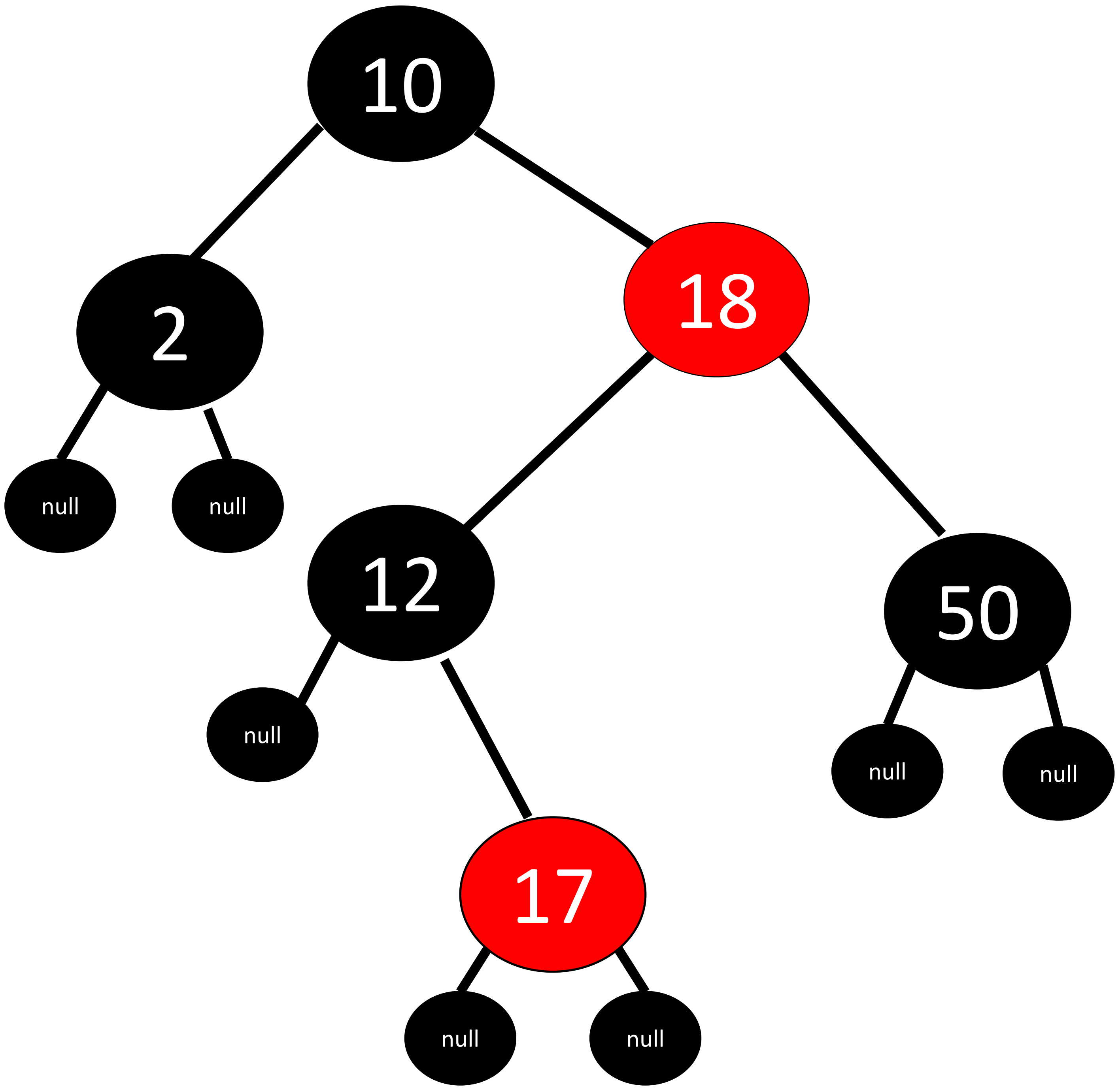
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

Path 1: 2 black nodes visited
Path 2: 2 black nodes visited
Path 3: 2 black nodes visited



Red-Black Tree Rules

5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes



Path 1: 2 black nodes visited

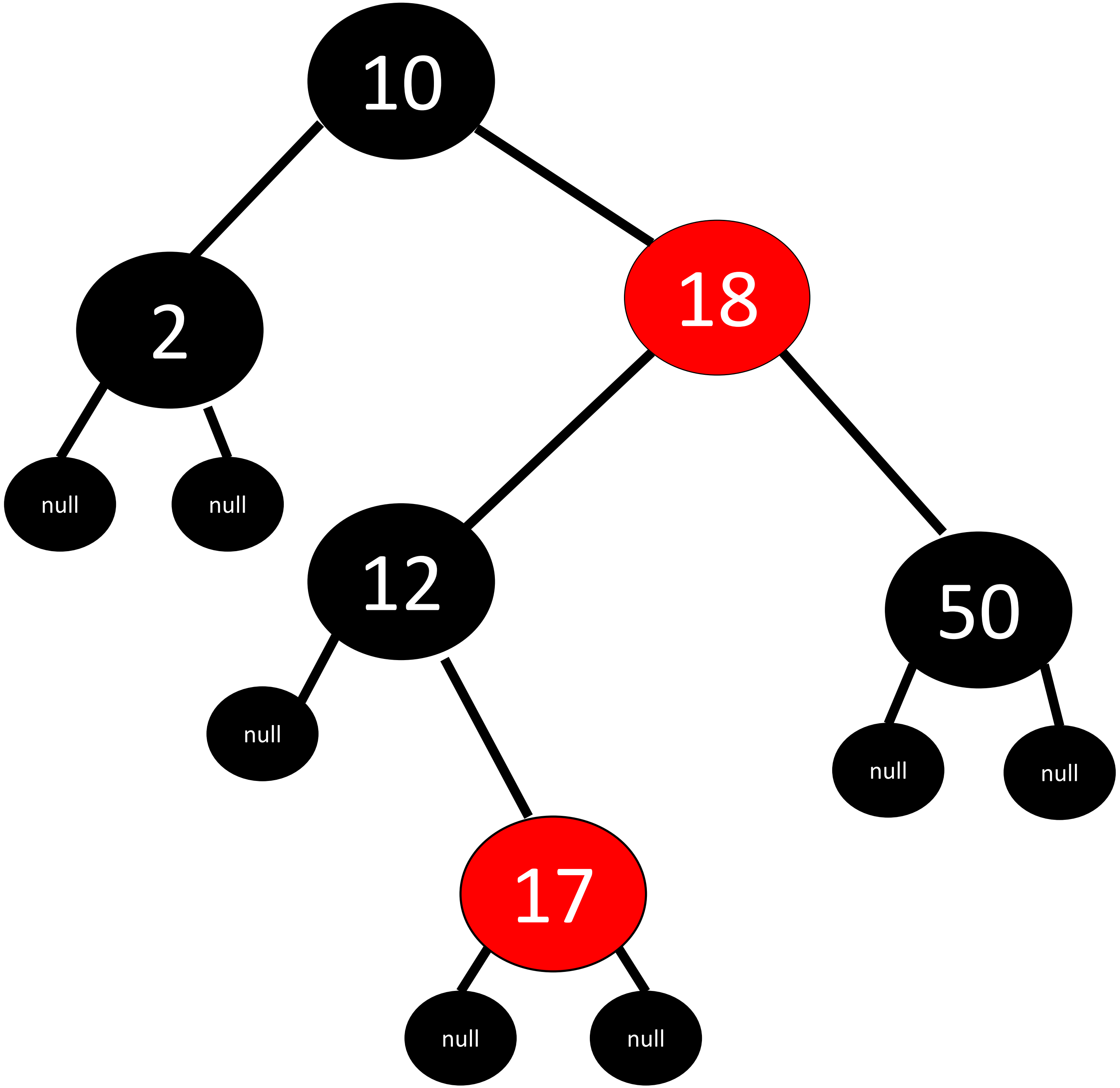
Path 2: 2 black nodes visited

Path 3: 2 black nodes visited

Red-Black Tree Rules

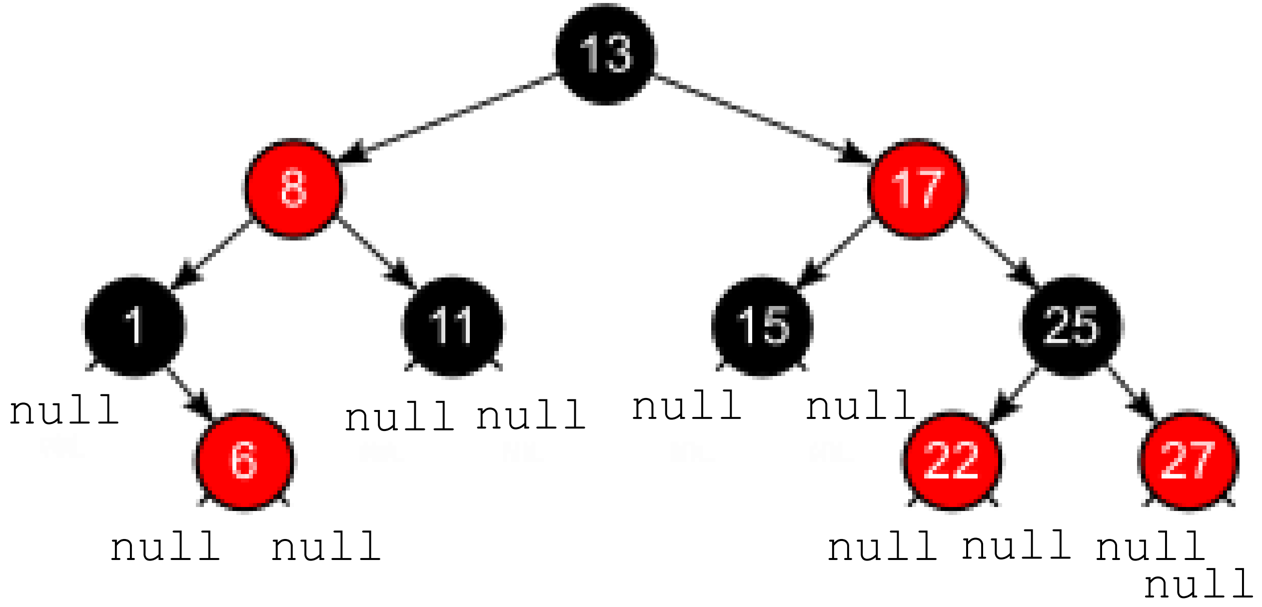
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

Path 1: 2 black nodes visited
Path 2: 2 black nodes visited
Path 3: 2 black nodes visited



Red-Black Tree Rules

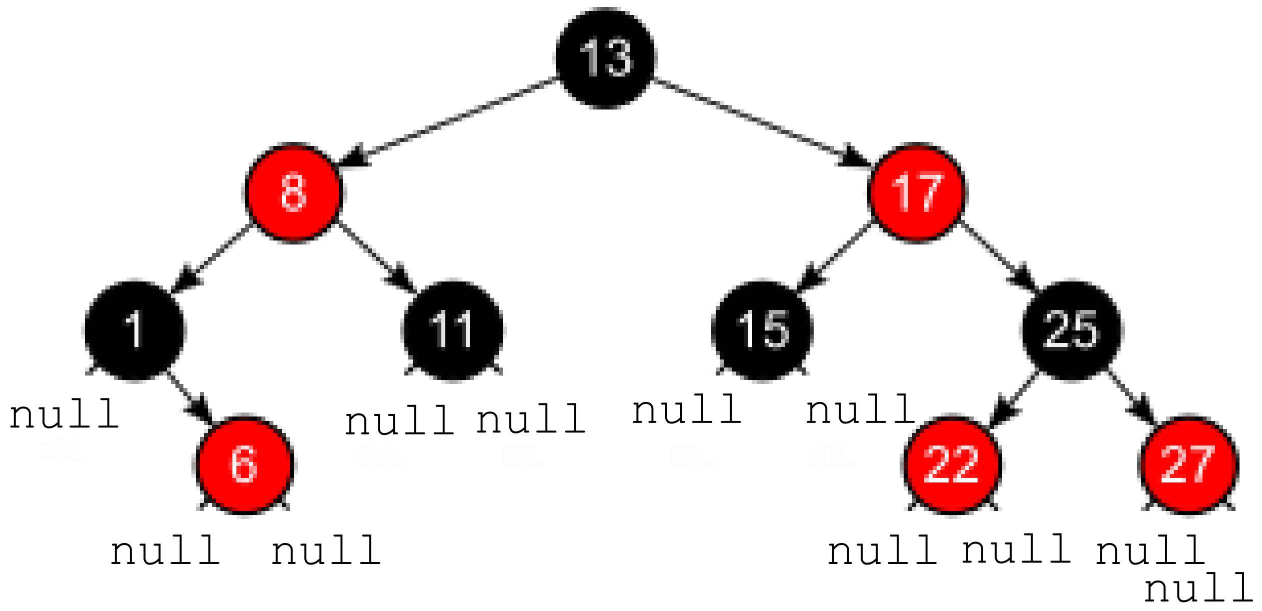
1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**
4. If a node is **red**, both children must be **black**
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes



Red-Black Tree Rules

1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**
4. If a node is **red**, both children must be **black**
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

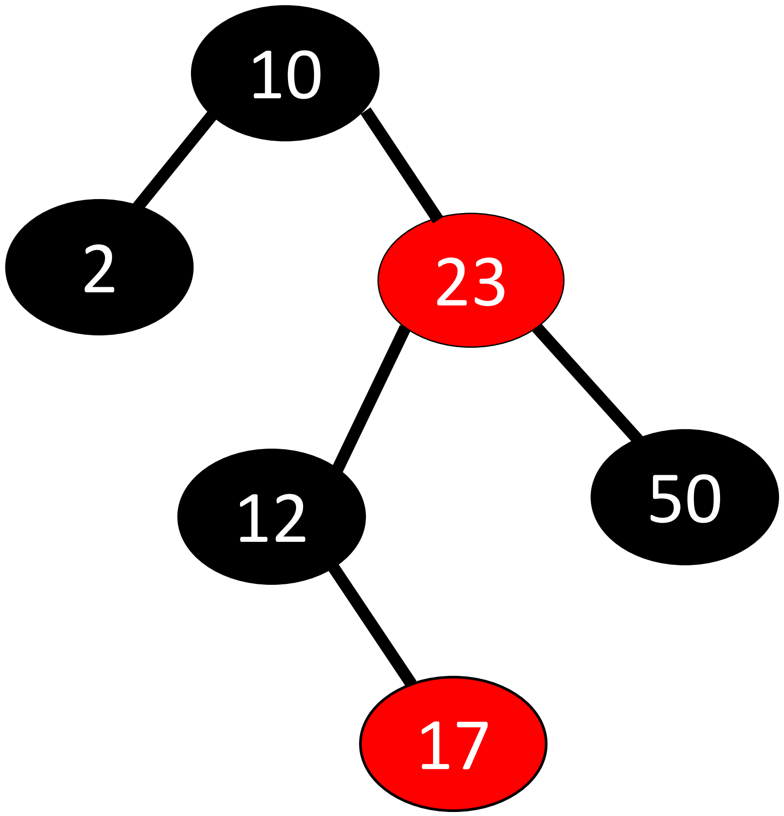
When we **insert** or **delete** something from a Red-Black tree, the new tree may **violate** one of these rules



Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion

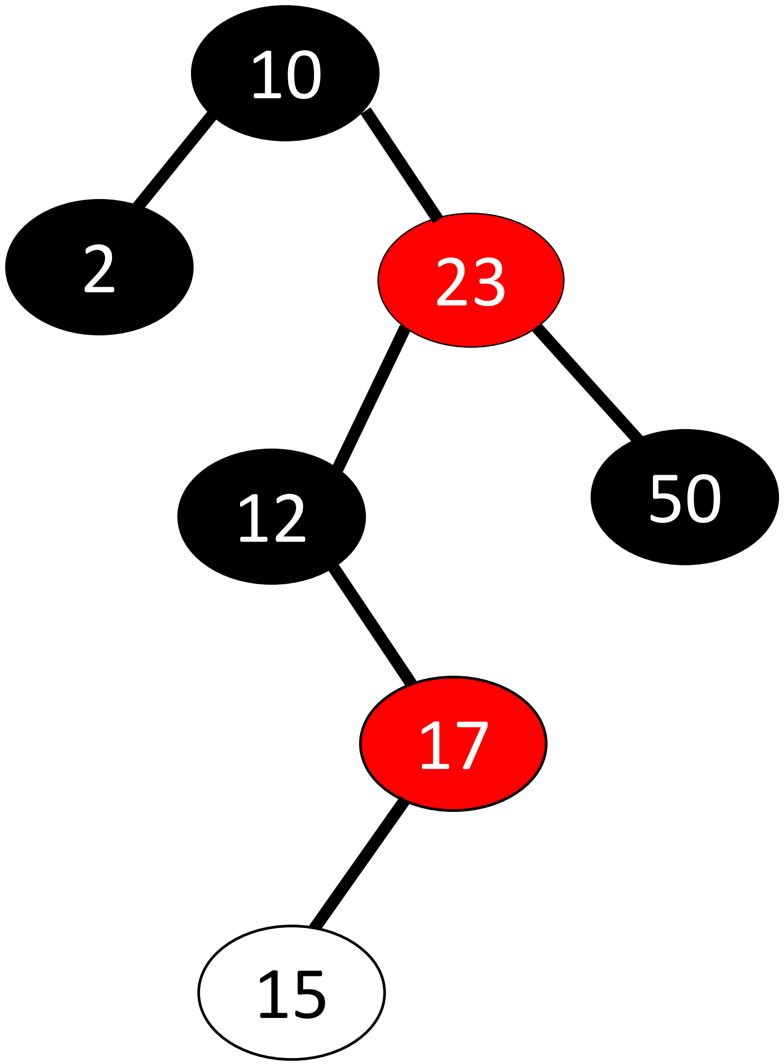


Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion

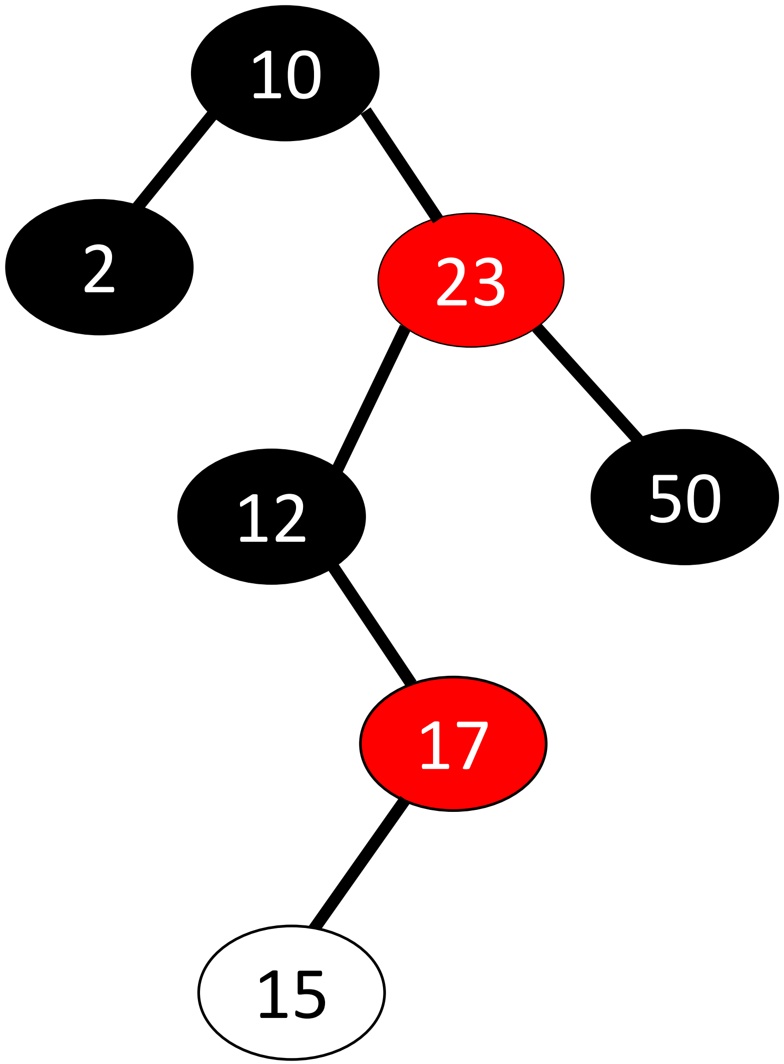
Our tree no longer has log(n) height, so we need to do some operations to reduce the height of the tree



Red-Black Tree Insertion/Deletion

`insert(15)`

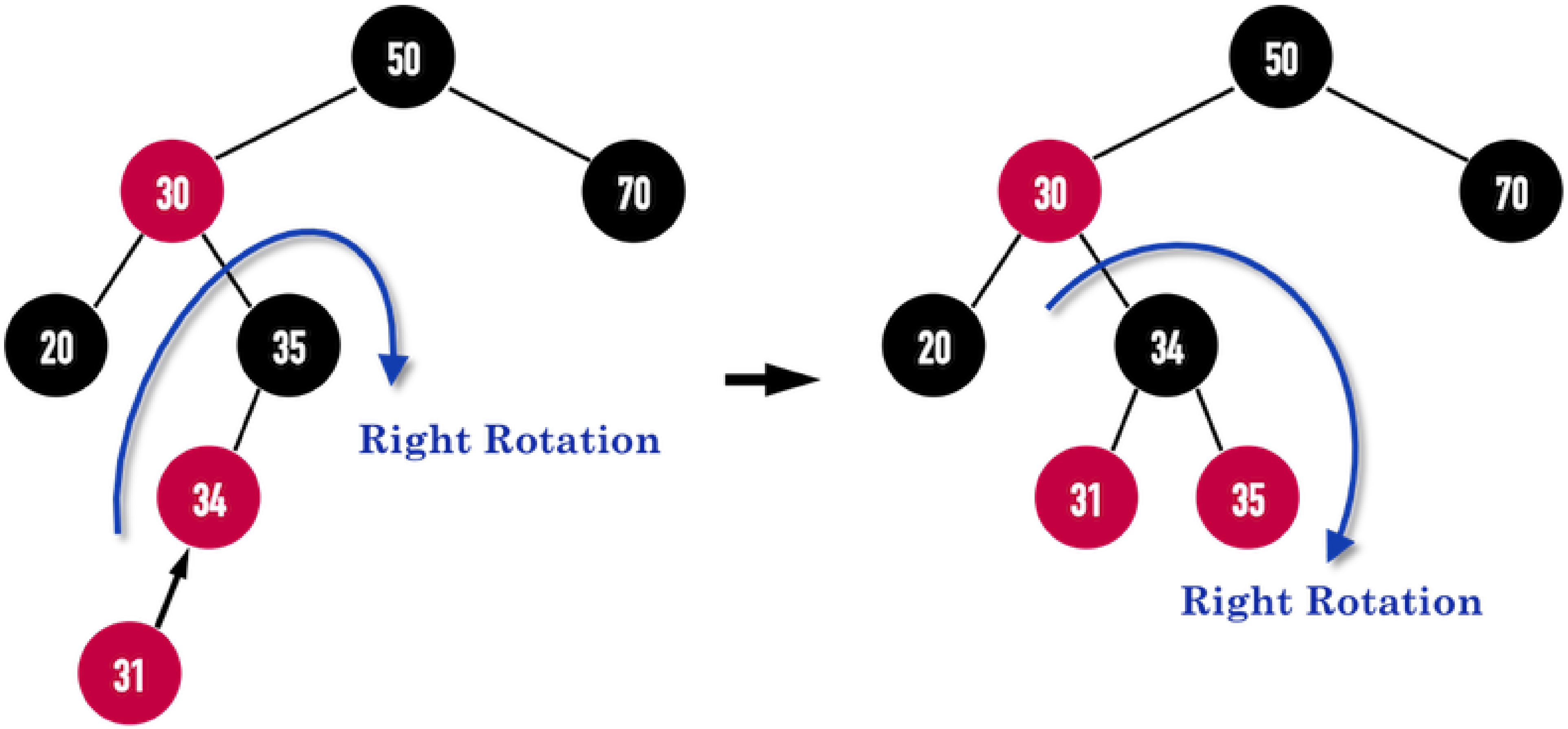
Step 1: Do the normal BST insertion



Our tree no longer has log(n) height, so we need to do some operations to reduce the height of the tree

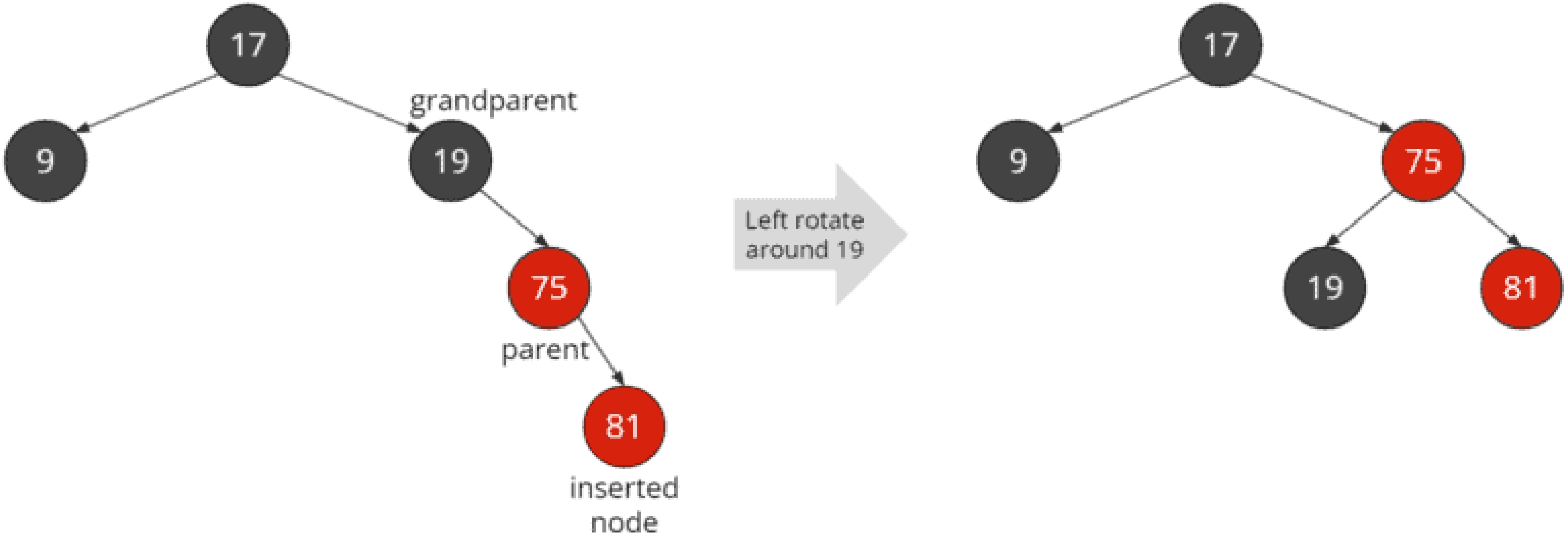
These operations are known as **rotations**

Red-Black Tree Rotation



Local transformation (we rotate just a section– not the entire tree)

Red-Black Tree Rotation

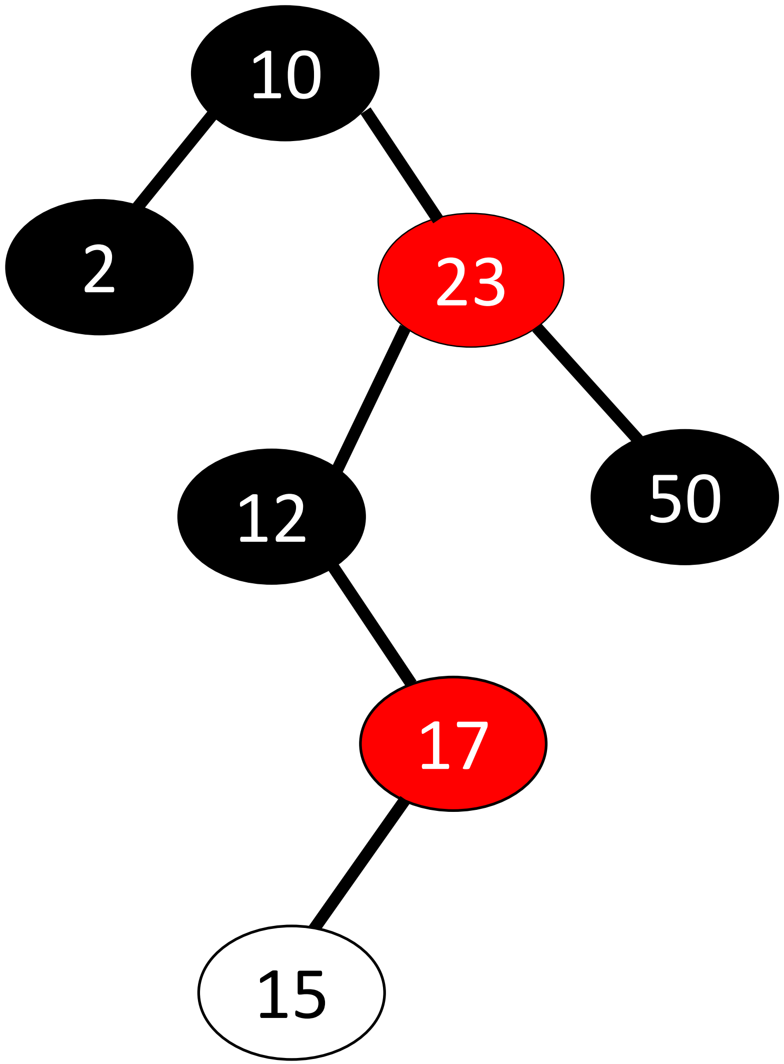


Local transformation (we rotate just a section– not the entire tree)

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion



Our tree no longer has log(n) height, so we need to do some operations to reduce the height of the tree

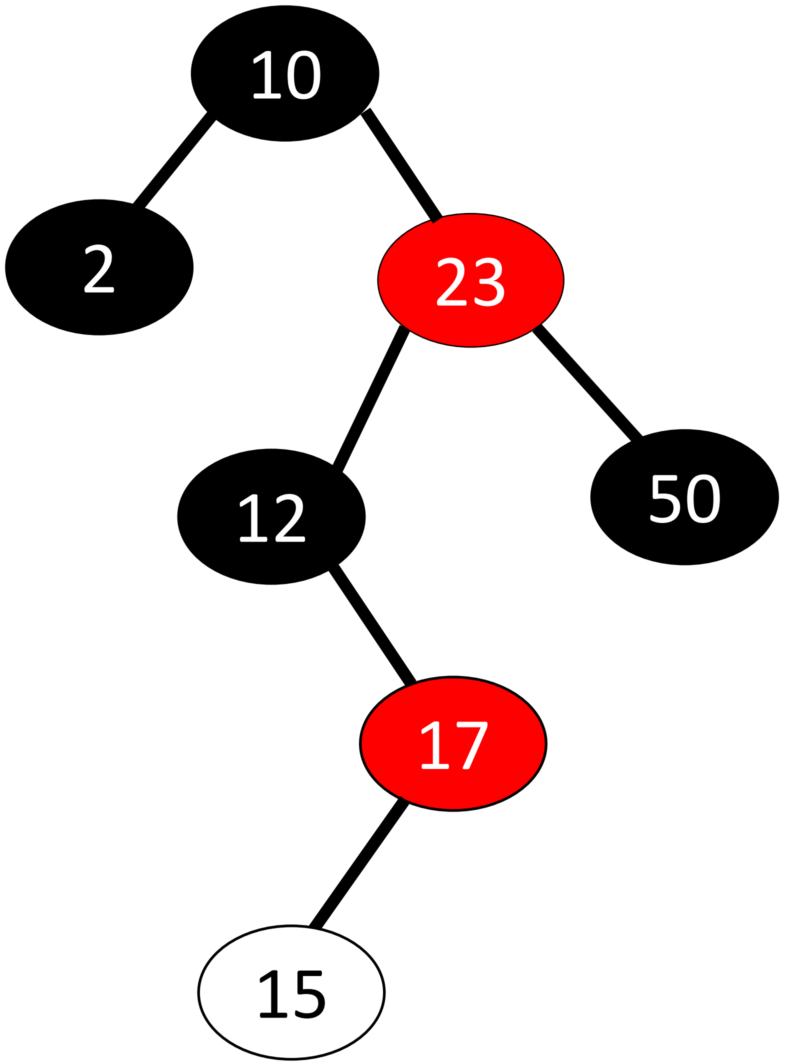
These operations are known as **rotations**

Red-Black Tree Insertion/Deletion

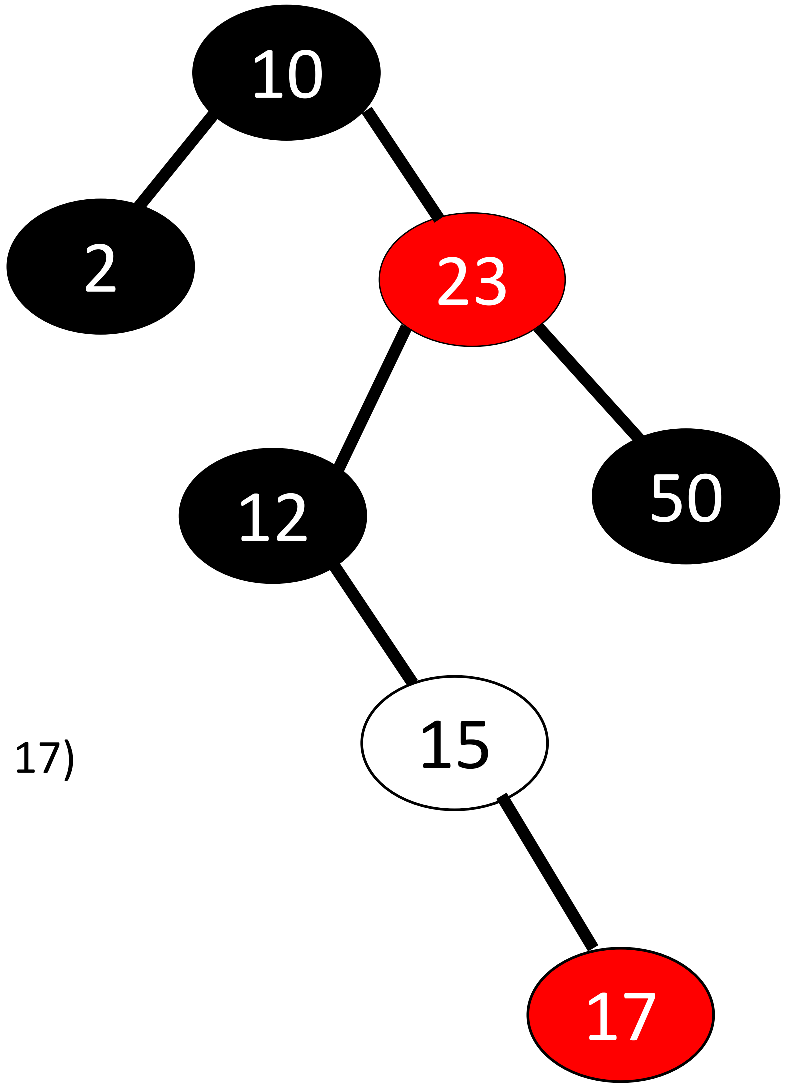
`insert(15)`

Step 1: Do the normal BST insertion

Step 2: Do rotation(s)



(Rotate Right around 17)

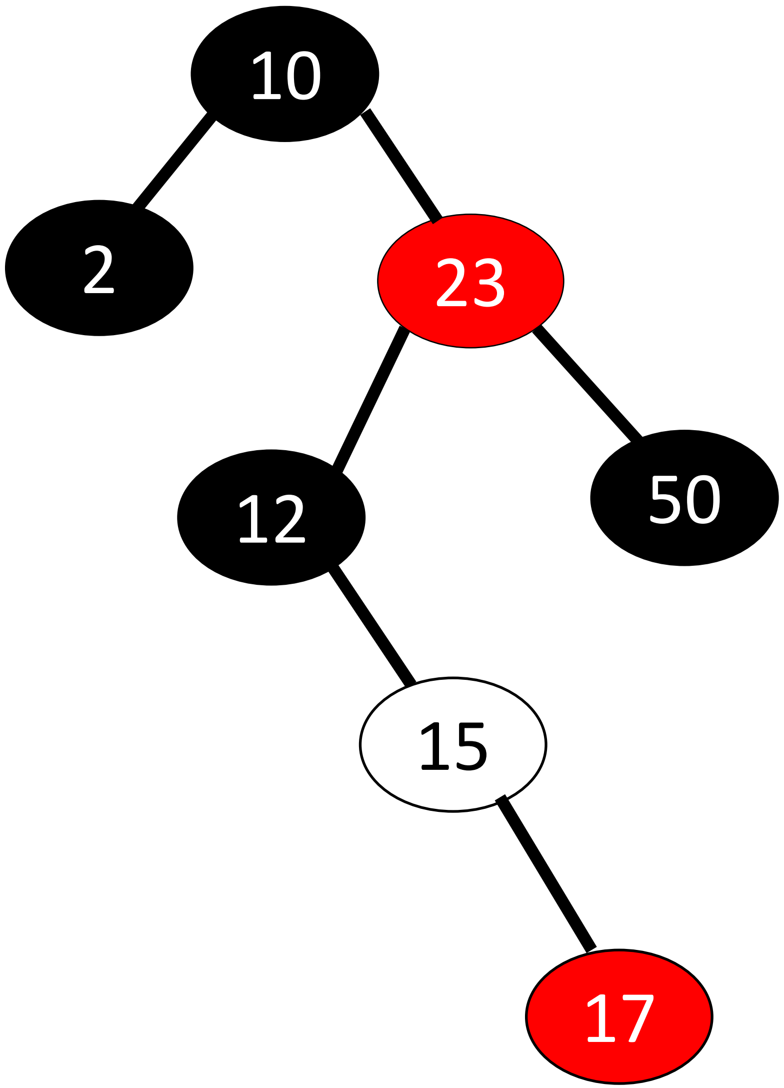


Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion

Step 2: Do rotation(s)

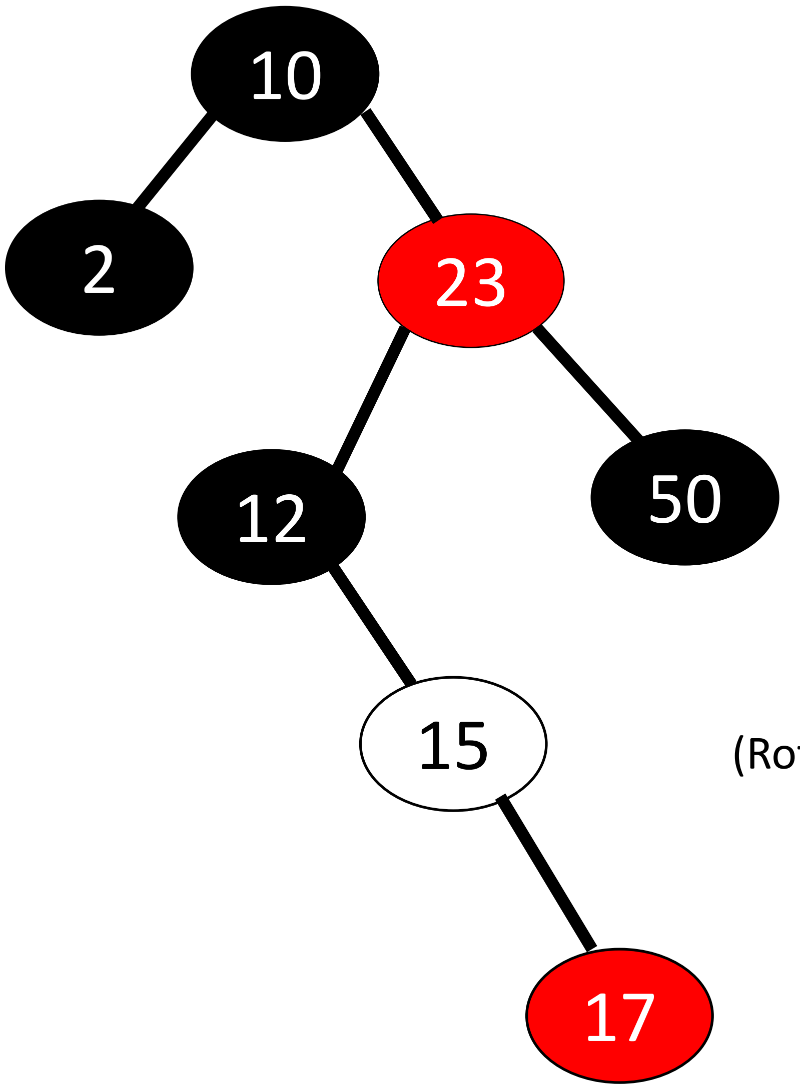


Red-Black Tree Insertion/Deletion

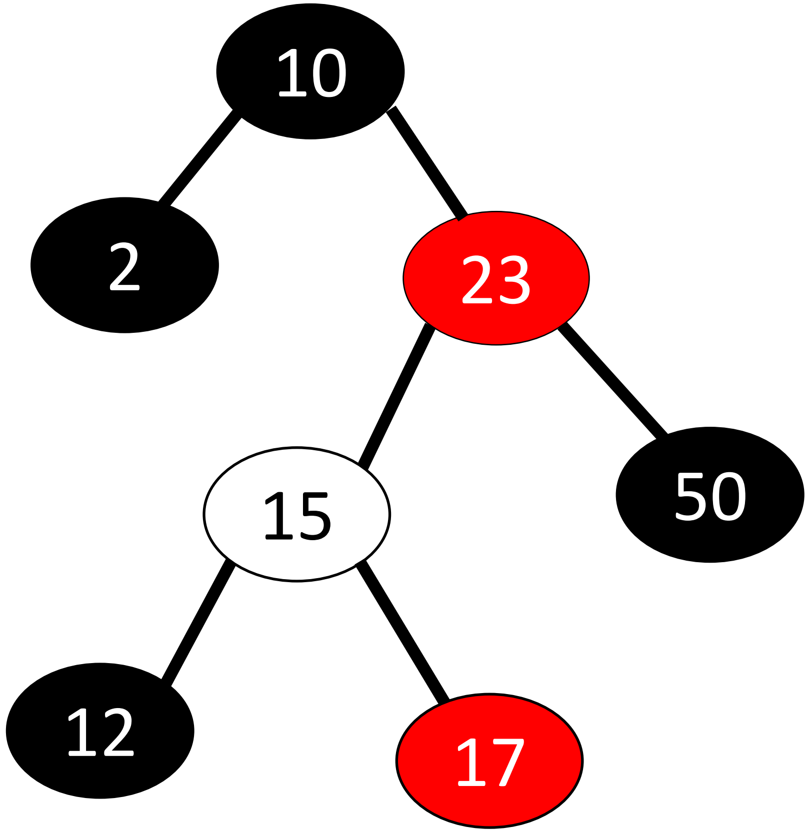
`insert(15)`

Step 1: Do the normal BST insertion

Step 2: Do rotation(s)



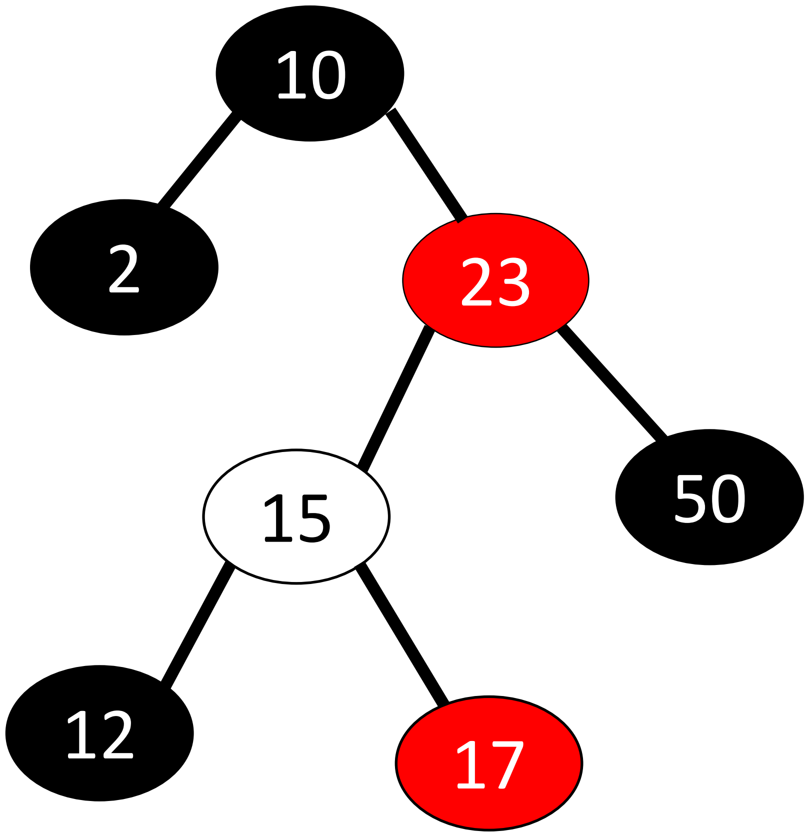
(Rotate left around 12)



Red-Black Tree Insertion/Deletion

`insert(15)`

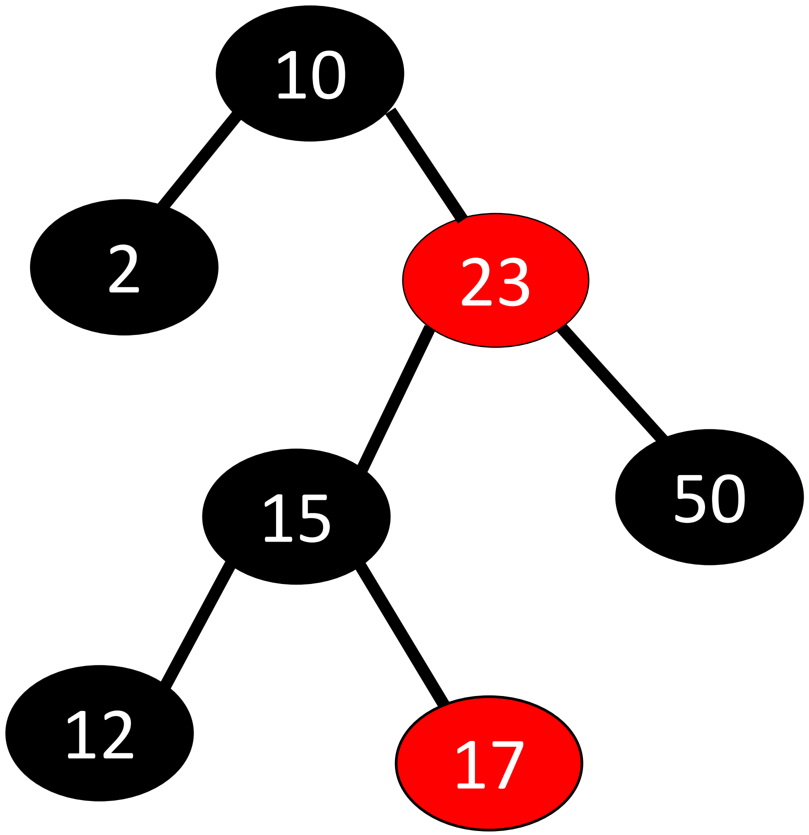
Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor



Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor

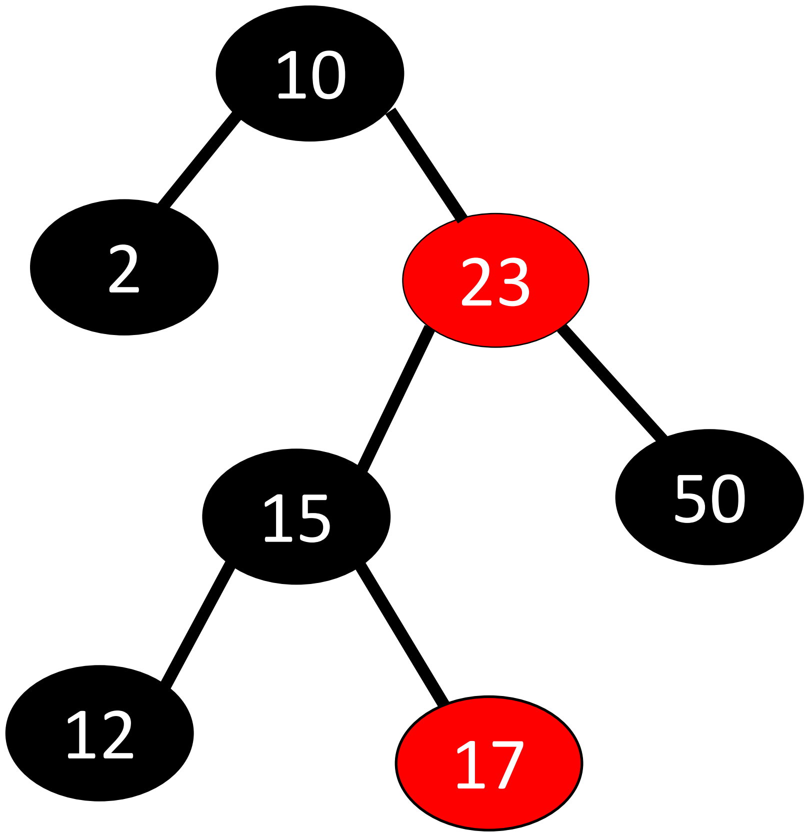


15 has to be black because….

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor

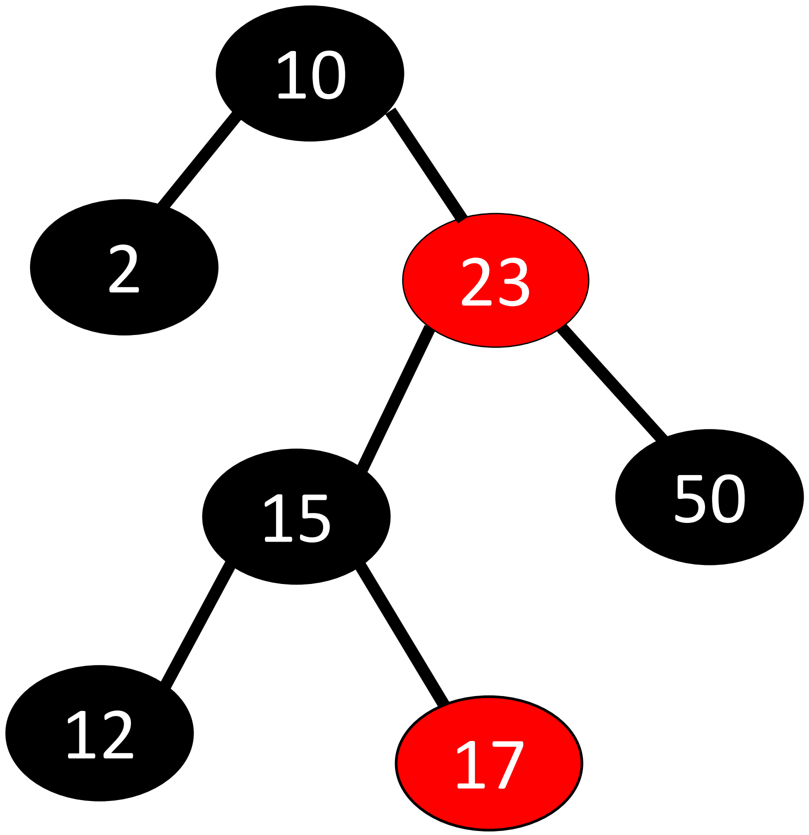


3. If a node is **red**, both children must be **black**

15 has to be black because 23 is red

Red-Black Tree Insertion/Deletion

`insert(15)`



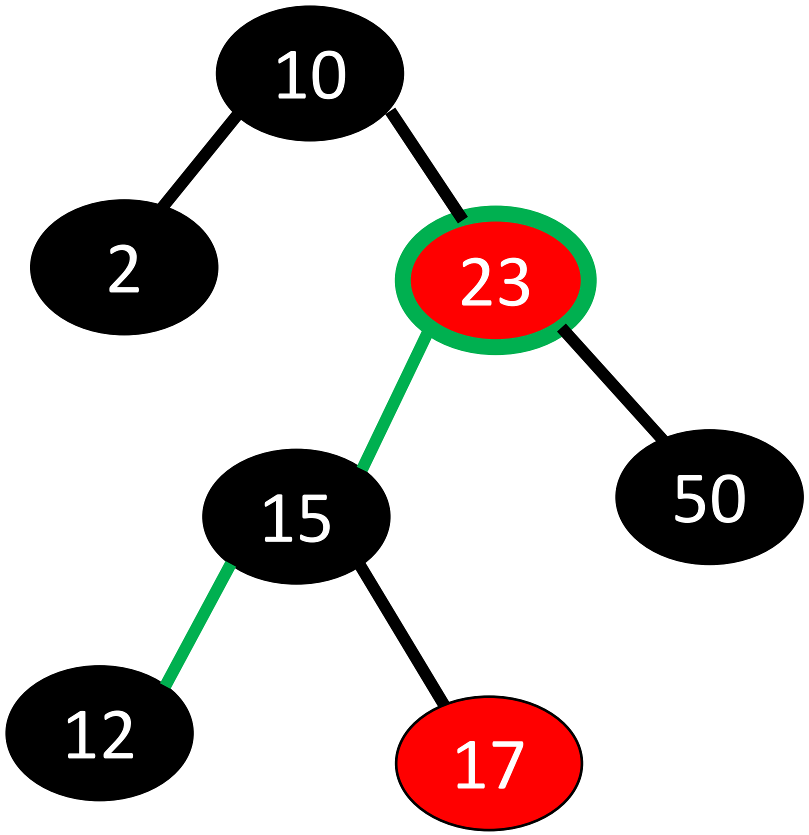
Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor

Is this a Red-Black tree?

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor



Is this a Red-Black tree?

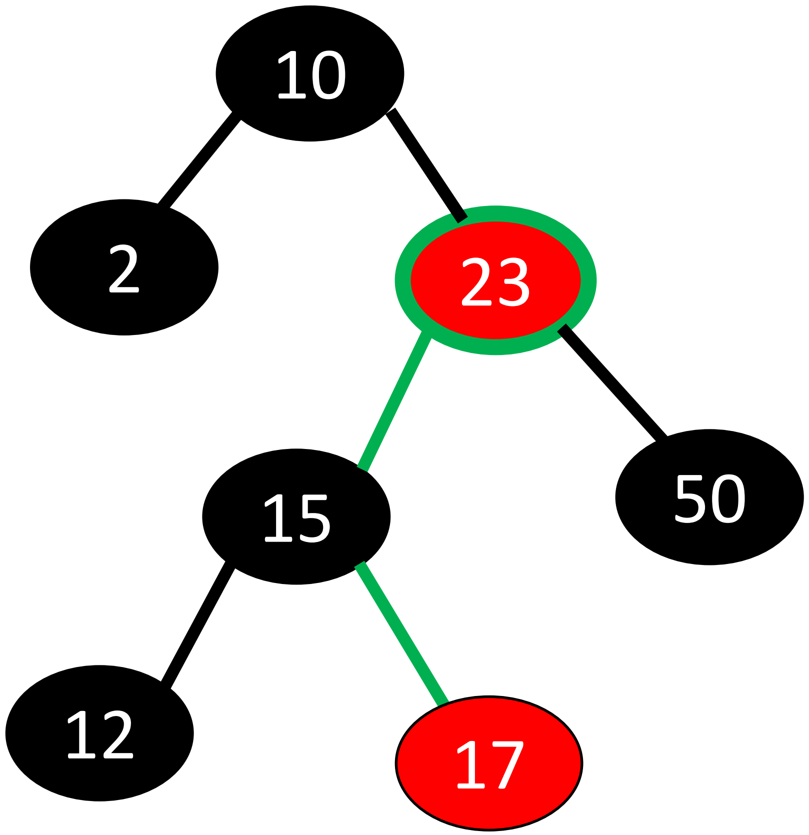
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

Path 1: 3 black nodes (including null nodes)

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor



Is this a Red-Black tree?

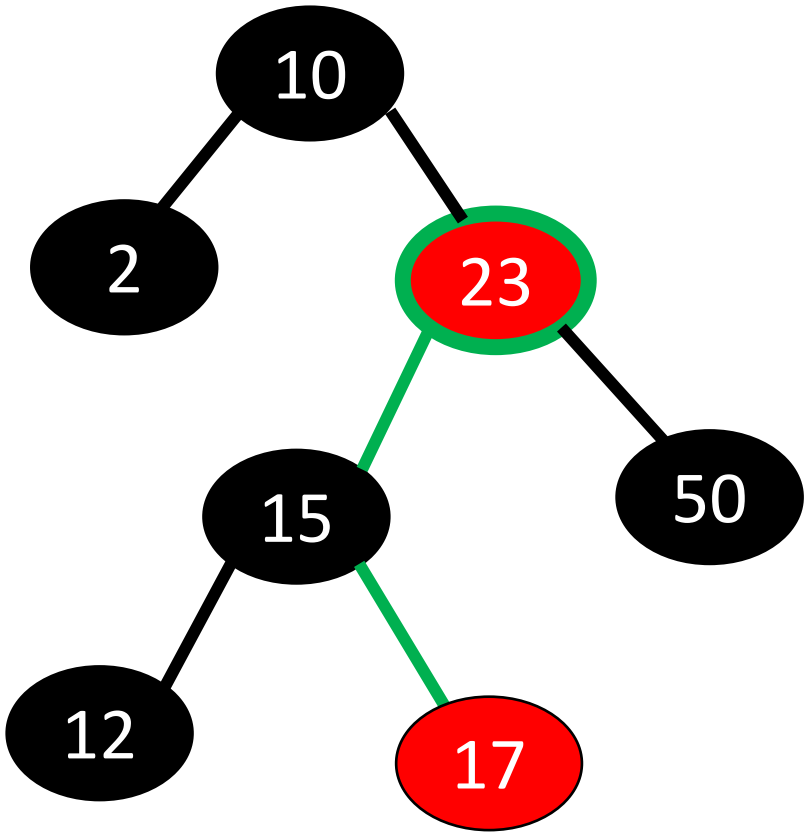
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

Path 1: 3 black nodes (including null nodes)
Path 2: 2 black nodes (including null nodes)

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor



Is this a Red-Black tree?

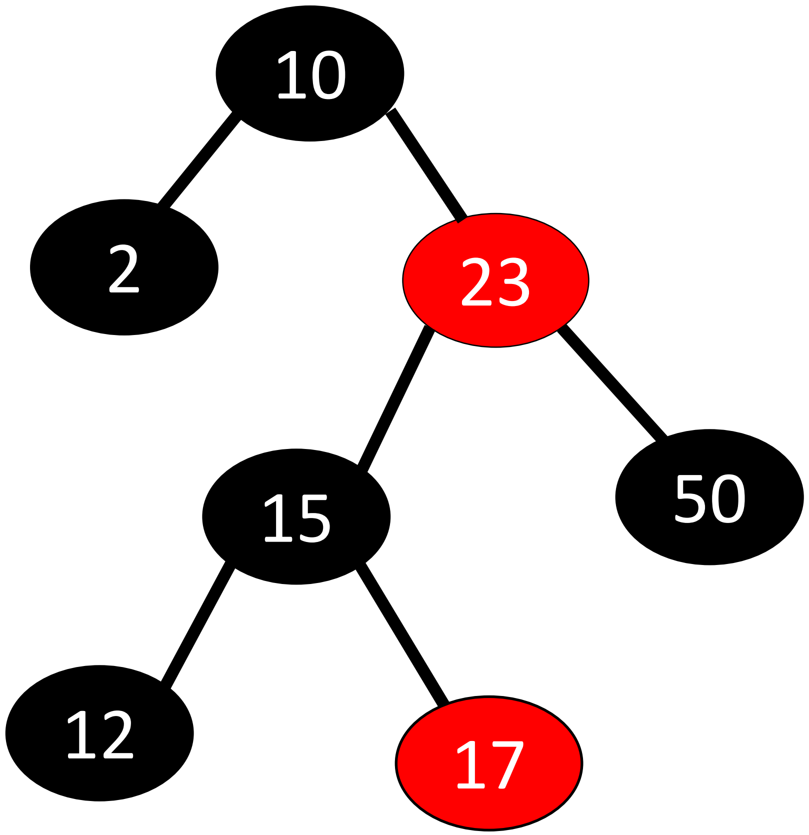
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

Path 1: 3 black nodes (including null nodes)
Path 2: 2 black nodes (including null nodes)

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor

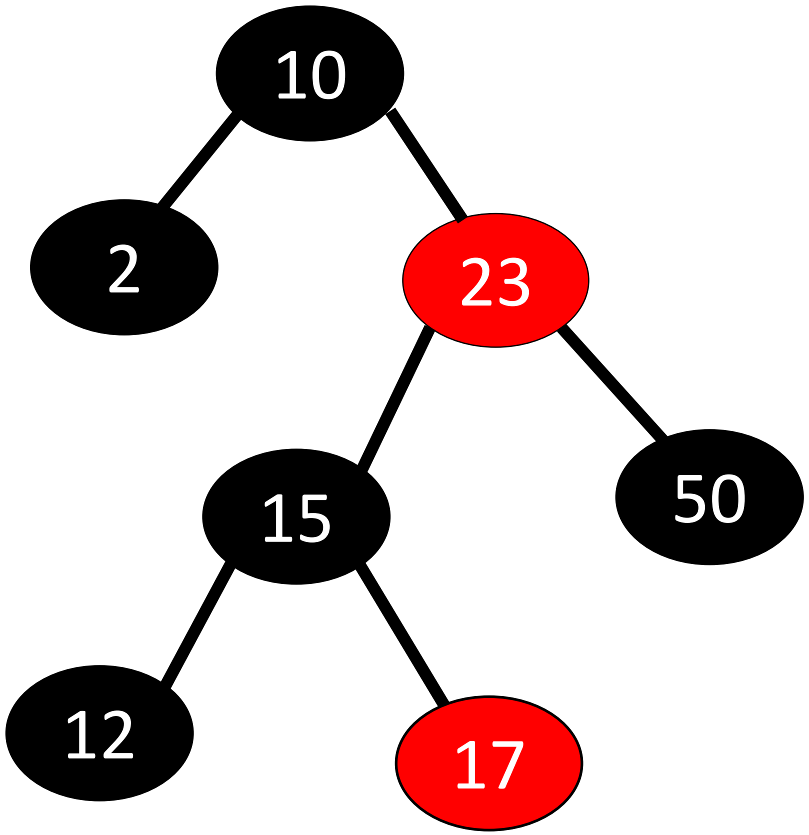


1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**
4. If a node is **red**, both children must be **black**
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor



Fact:

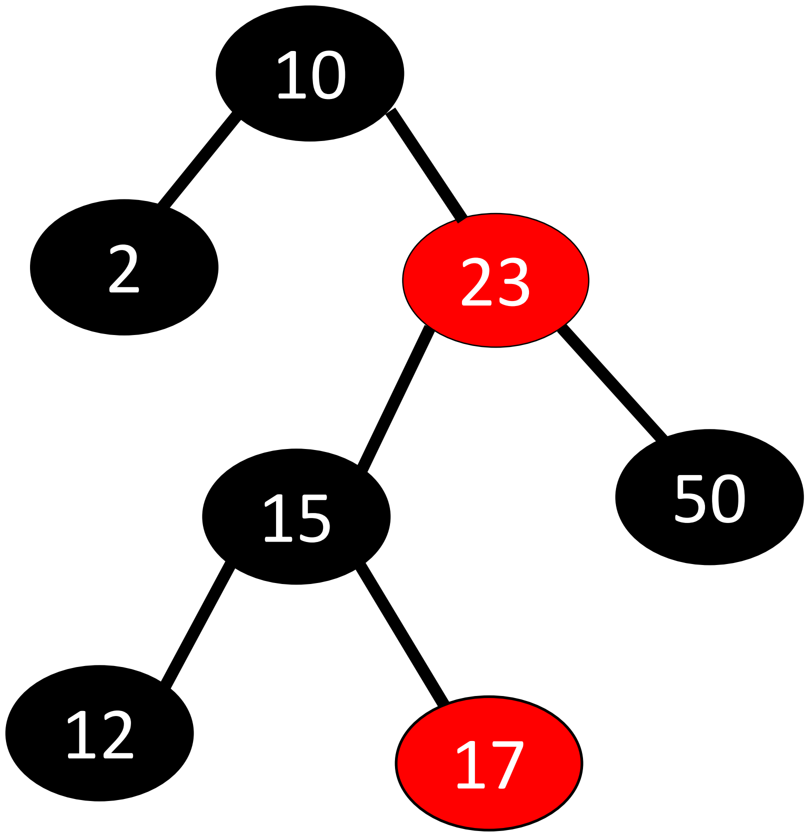
There will at most 3 rotations needed, and each rotation happens in O(1) time

So, maintaining a Red/Black try happens in O(1) time

Red-Black Tree Insertion/Deletion

`delete(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor



Fact:

There will at most 3 rotations needed, and each rotation happens in O(1) time

So, maintaining a Red/Black try happens in O(1) time

Red-Black Tree Insertion/Deletion

`delete(15)`

(Deleting is not as scary, because deleting a node will never increase the height of the tree)

Step 1: Do the normal BST deletion

- Case 1: no children
- Case 2: 1 child
- Case 3: 2 children

Step 2: Do rotation(s) (optional?)

Step 3: Recolor

Fact:

There will at most 3 rotations needed, and each rotation happens in O(1) time

So, maintaining a Red/Black try happens in O(1) time

Takeaways

We can add a color (red or black) instance field to our nodes to create a Red Black Tree

If we follow the rules of a Red Black Tree, and follow the proper rotations/recoloring steps, we can guarantee that our tree will be balanced

Guaranteed Balanced BST =
- ❑ O(logn) insertion
- ❑ O(logn) deletion
- ❑ O(logn) Searching

There are also BSTs called **AVL tree** and **2-3 trees** that serve the same purpose of RB trees

	Array	**Linked List**	**BST (Balanced)**
Insertion	O(n)	O(1)	O(logn)
Deletion	O(n)	O(n)**	O(logn)
Searching	O(logn)*	O(n)	O(logn)

*if array is sorted

**unless we are removing the head or tail