CSCI 232:
Data Structures and Algorithms

Red Black Trees

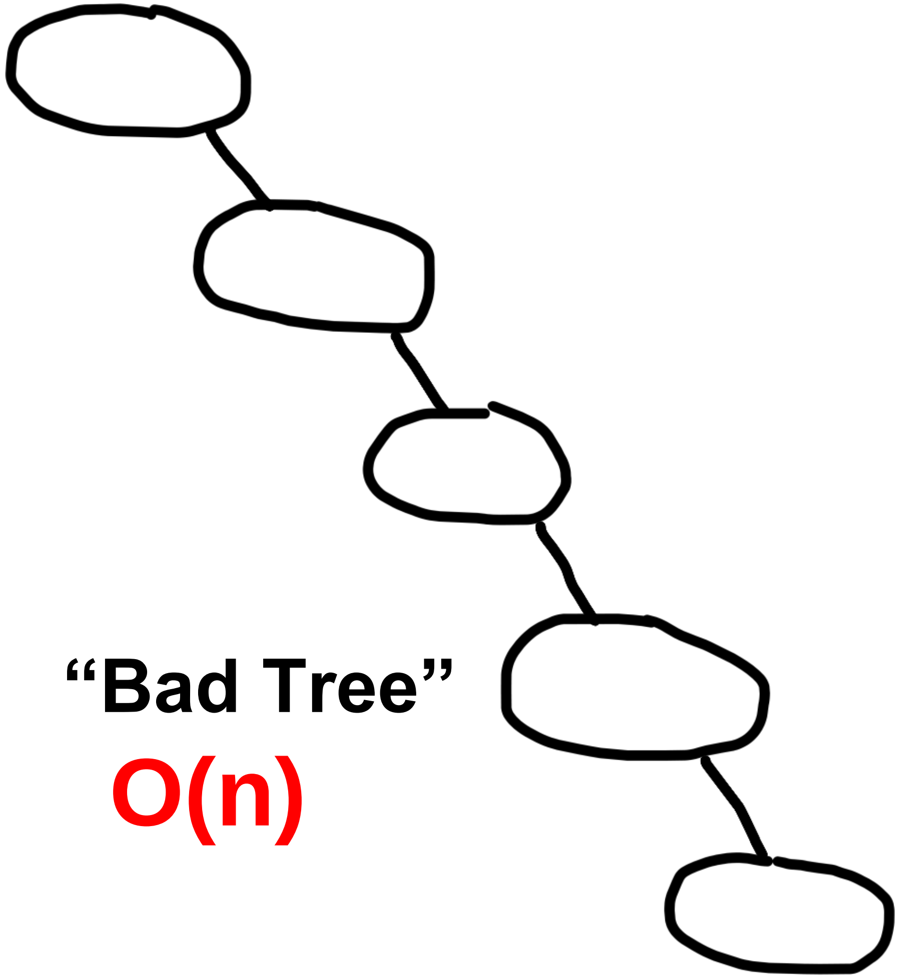
Reese Pearsall
Spring 2025

Announcements

Quiz 1 due tomorrow!!
(No lab this week, but you **must** go to lab tomorrow to do the quiz)

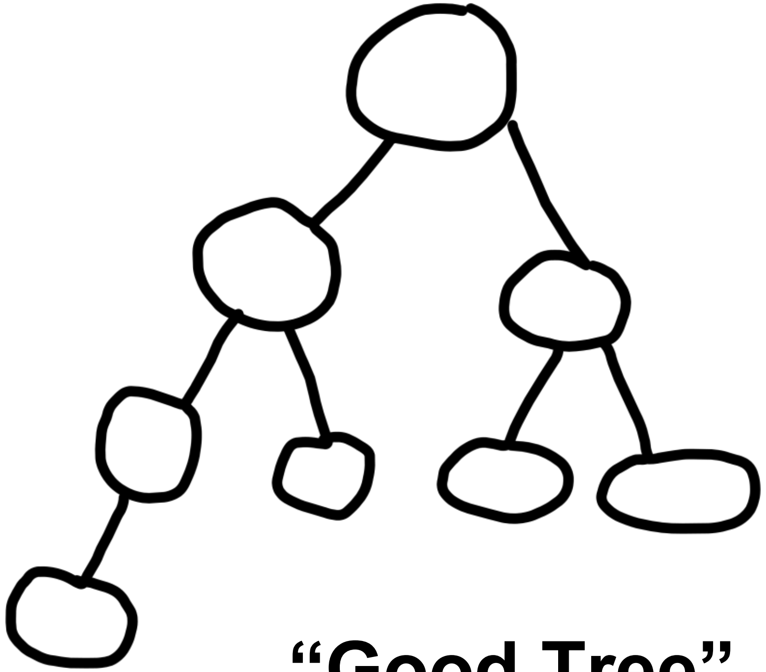
Binary Search Tree – Insertion/Searching/Removing

Running time?



"Bad Tree"

O(n)

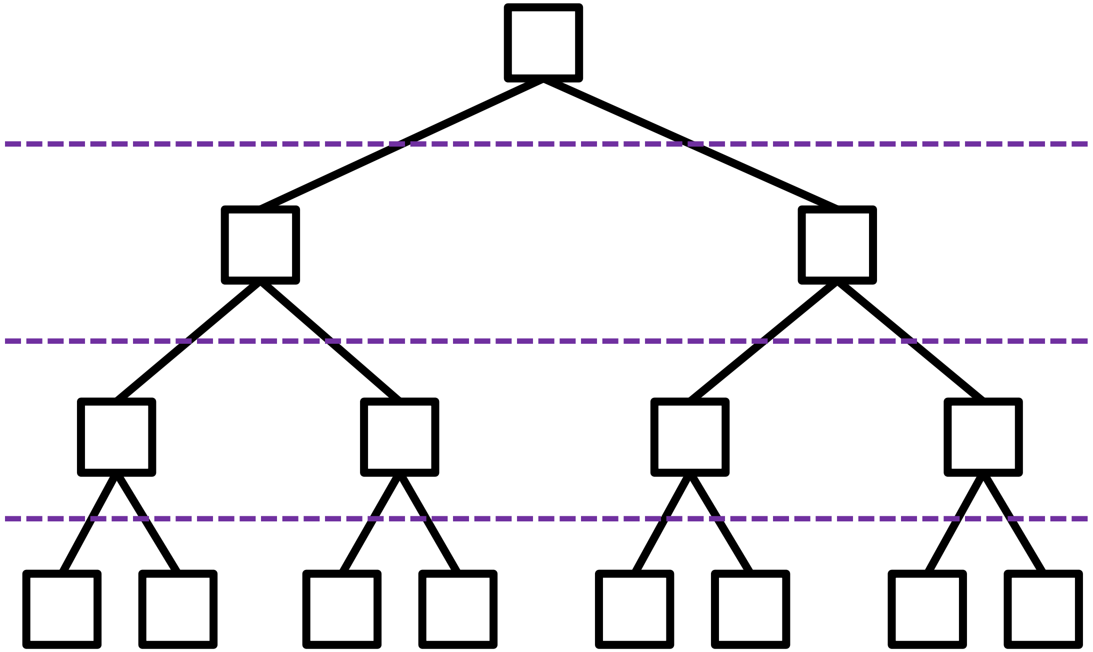


"Good Tree"

O(logn)

Balanced BST

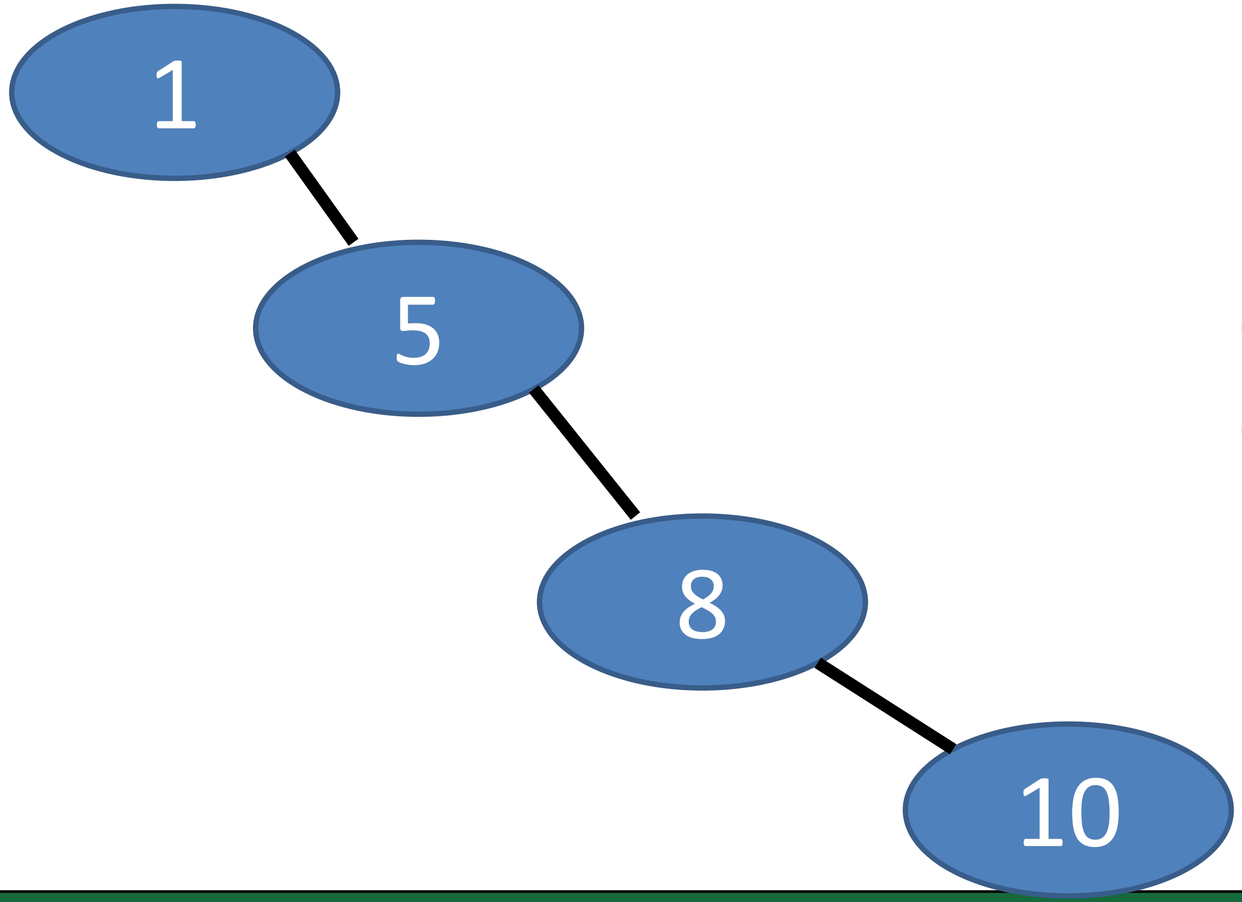
A **balanced** binary tree, is defined as a binary tree in which given n nodes, the height of the tree is `O(logn)`.



Depth	Num Nodes
0	1
1	2
2	4
3	8

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)`.

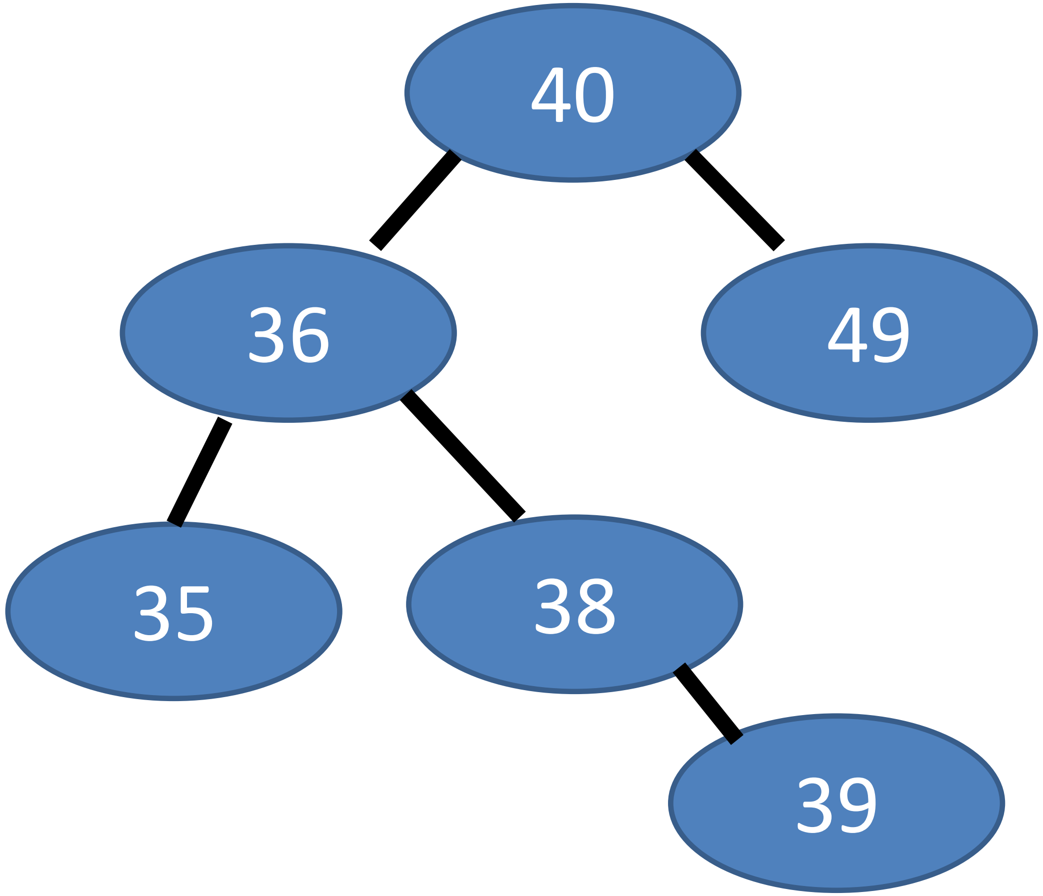


4 nodes
→ If this is a balanced tree, the height should be less than or equal to 2 (log(4))

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)**.

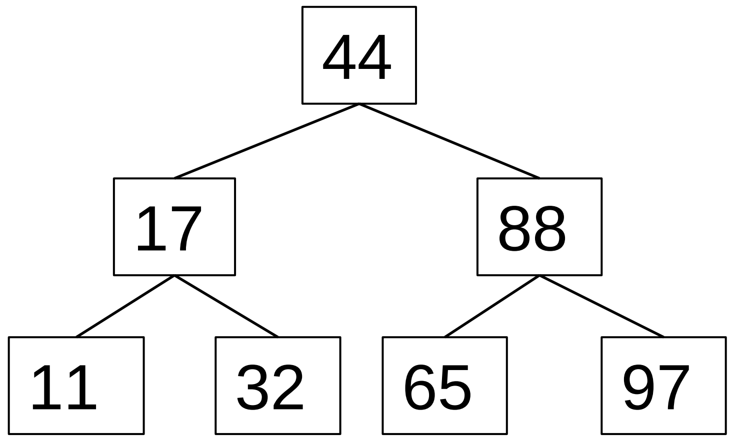


6 nodes
→ If this is a balanced tree, the height should be less than or equal to 3
ceil(log(6))

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)



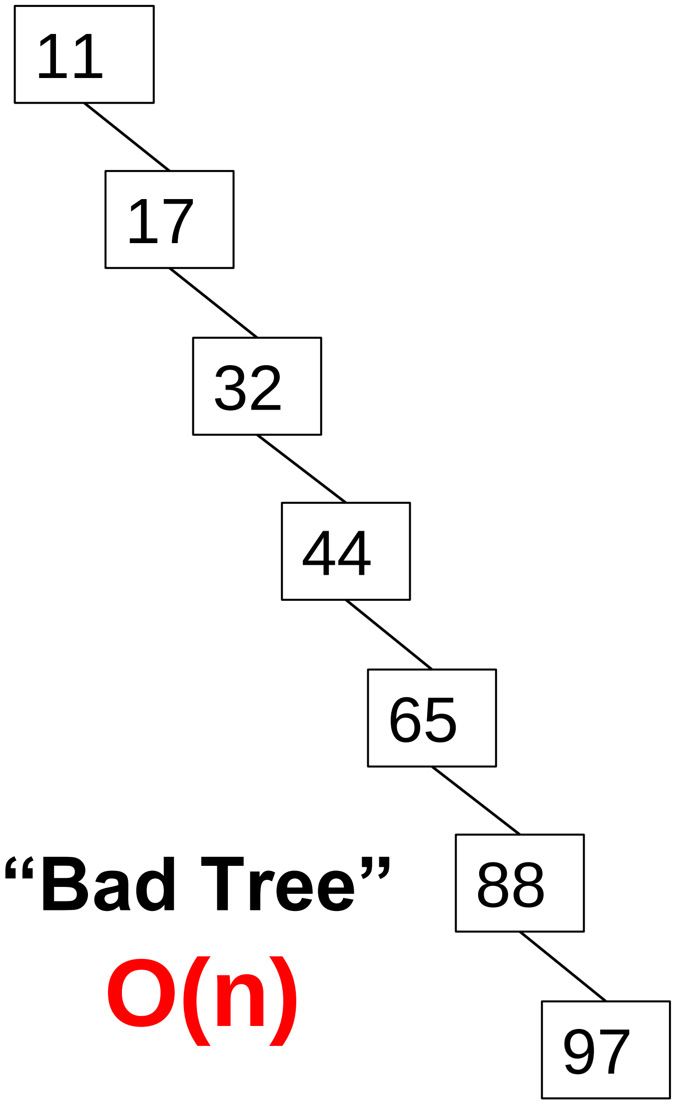
44, 17, 88, 11, 32, 65, 97

44, 17, 32, 88, 11, 97, 65

44, 88, 65, 97, 17, 32, 11

“Good Tree”

O(logn)



“Bad Tree”

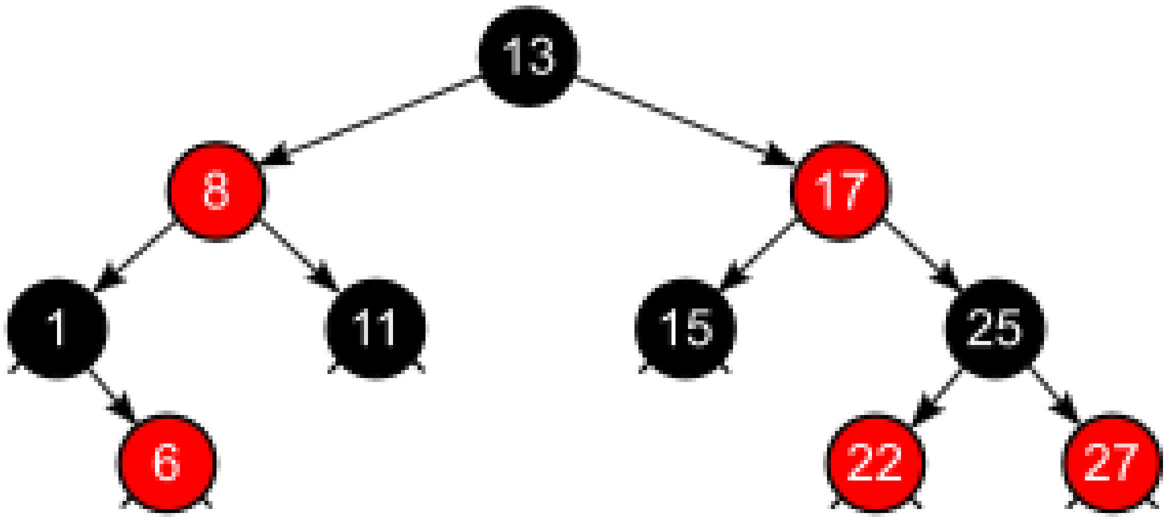
O(n)

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

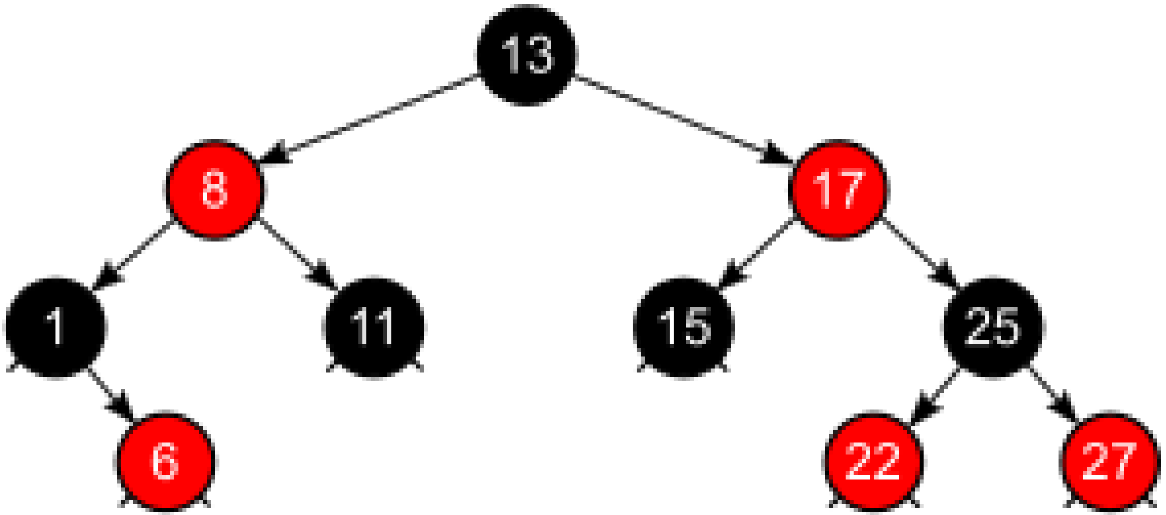


Balanced BST

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 greater than the node
- A node cannot have more than two children
- No duplicate nodes

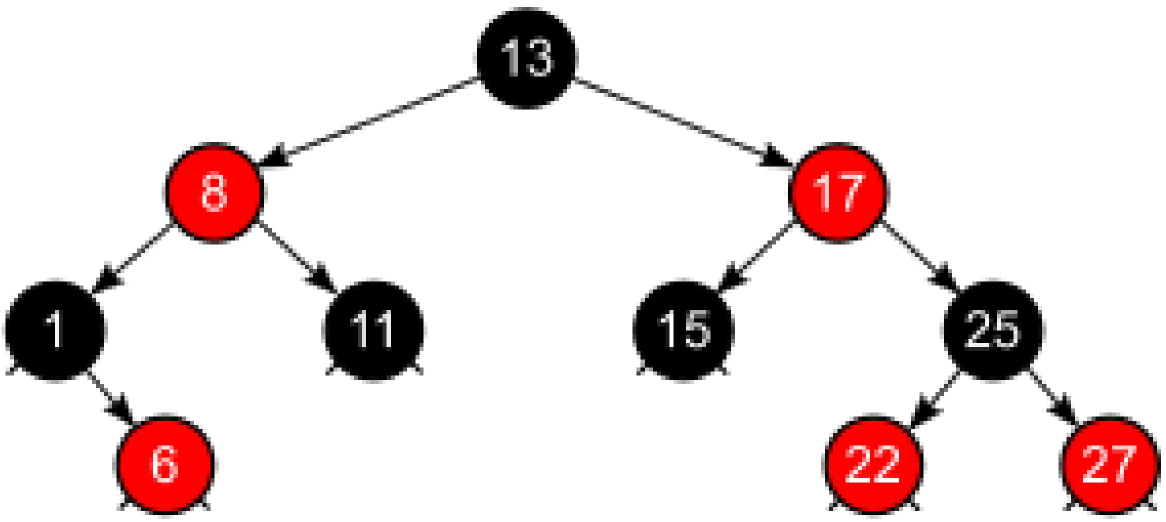
(BST Rules)



Red-Black Tree Rules

Each Node now has a **color** (red or black)

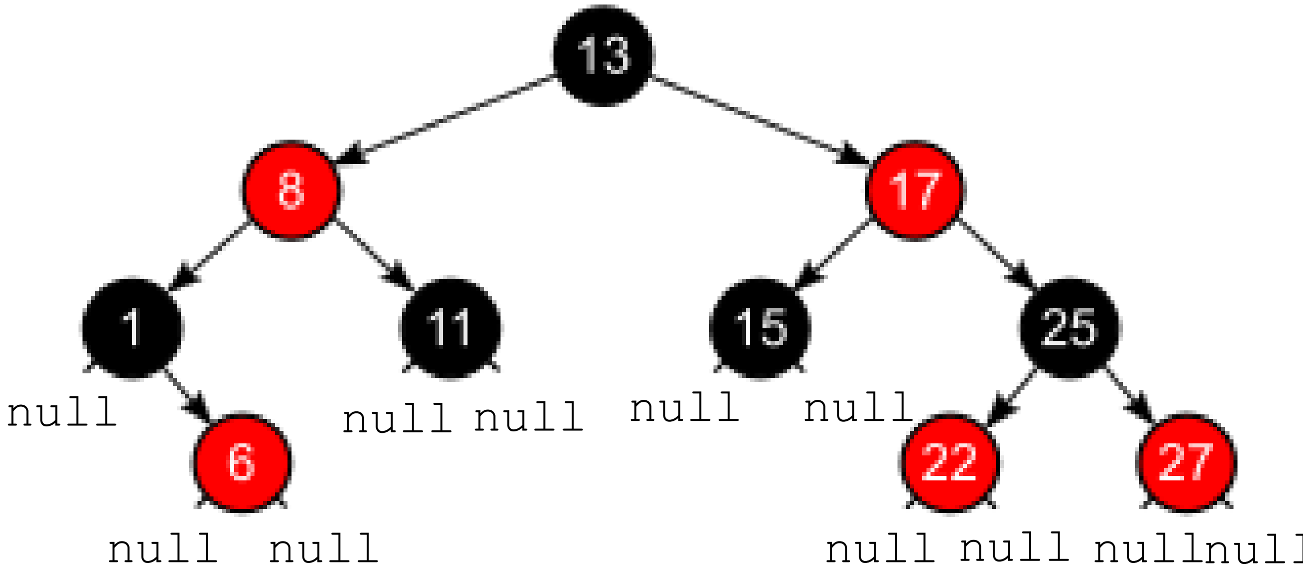
1. Every node is either **red** or **black**



Red-Black Tree Rules

Each Node now has a **color** (red or black)

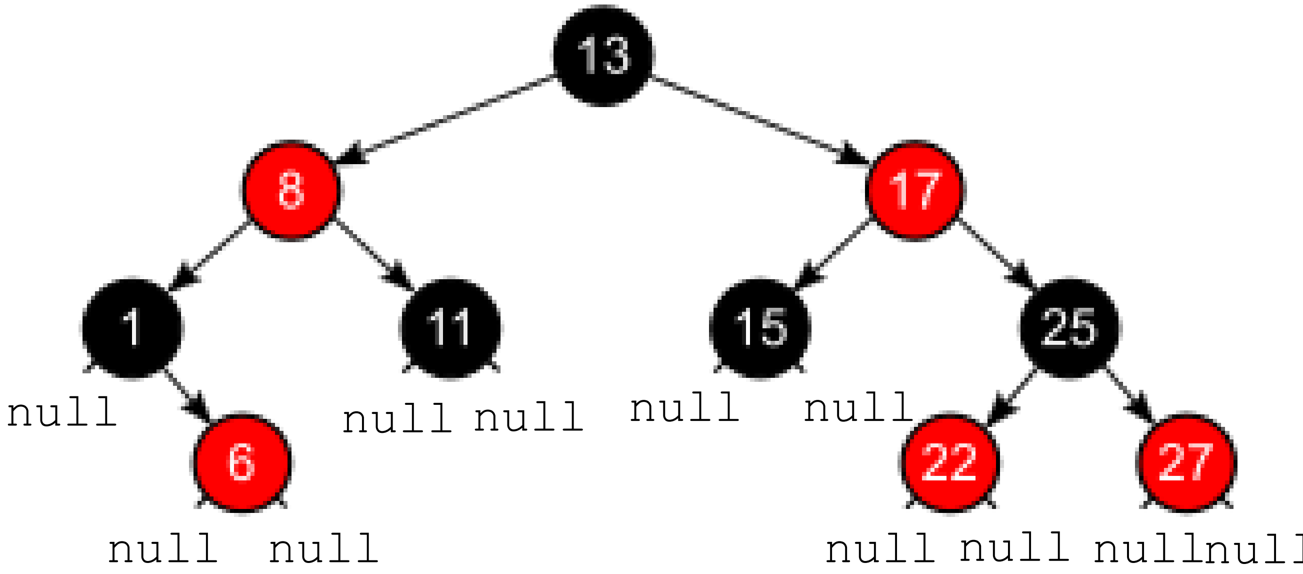
1. Every node is either **red** or **black**
2. The `null` children are **black**



Red-Black Tree Rules

Each Node now has a **color** (red or black)

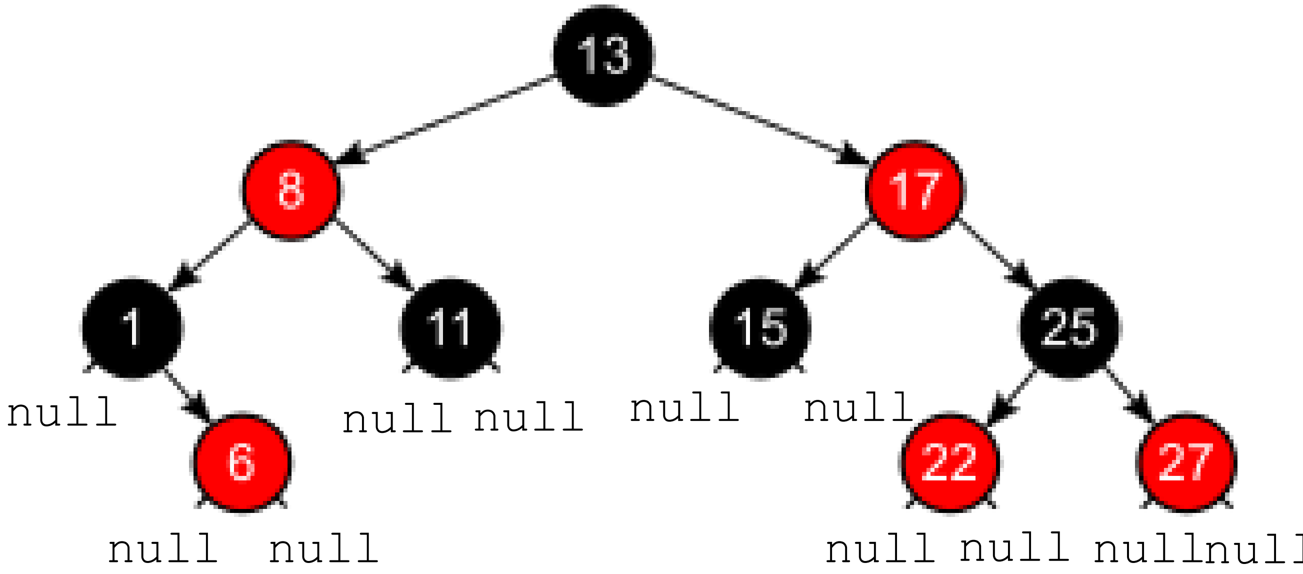
1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**



Red-Black Tree Rules

Each Node now has a **color** (red or black)

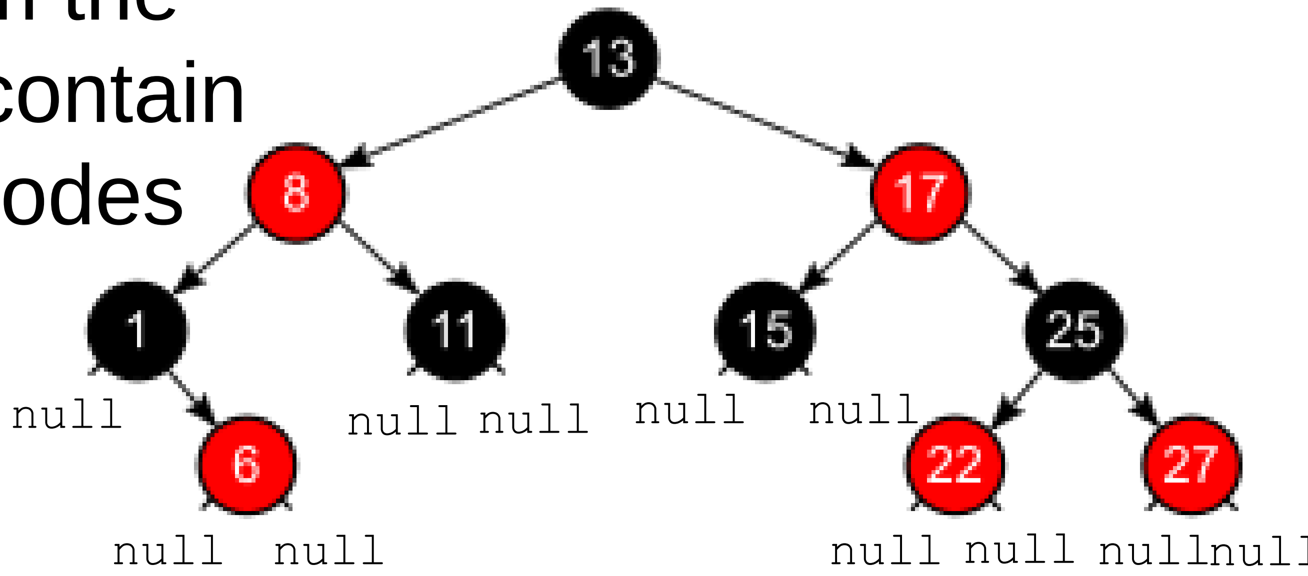
1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**
4. If a node is **red**, both children must be **black**



Red-Black Tree Rules

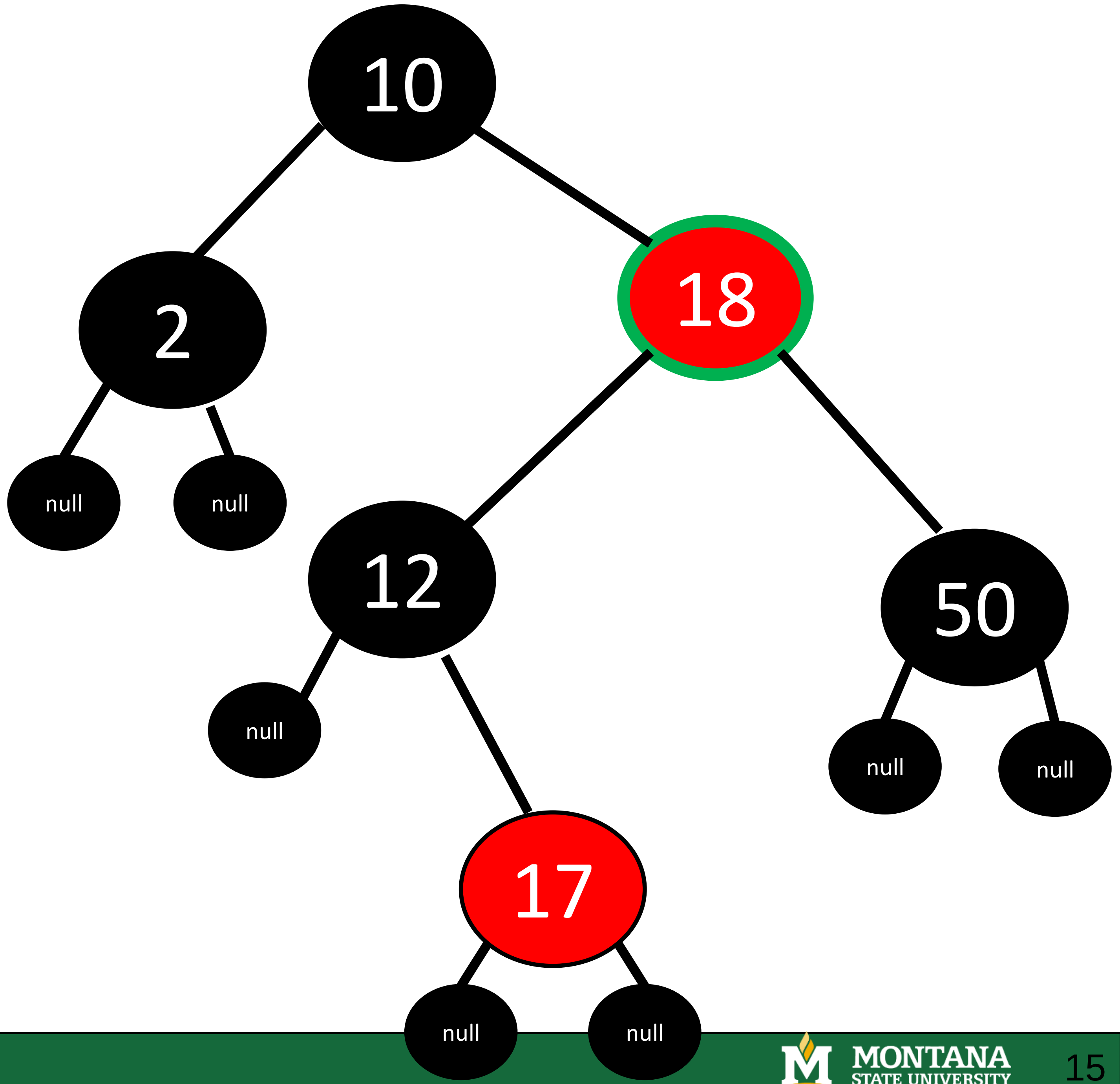
Each Node now has a **color** (red or black)

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

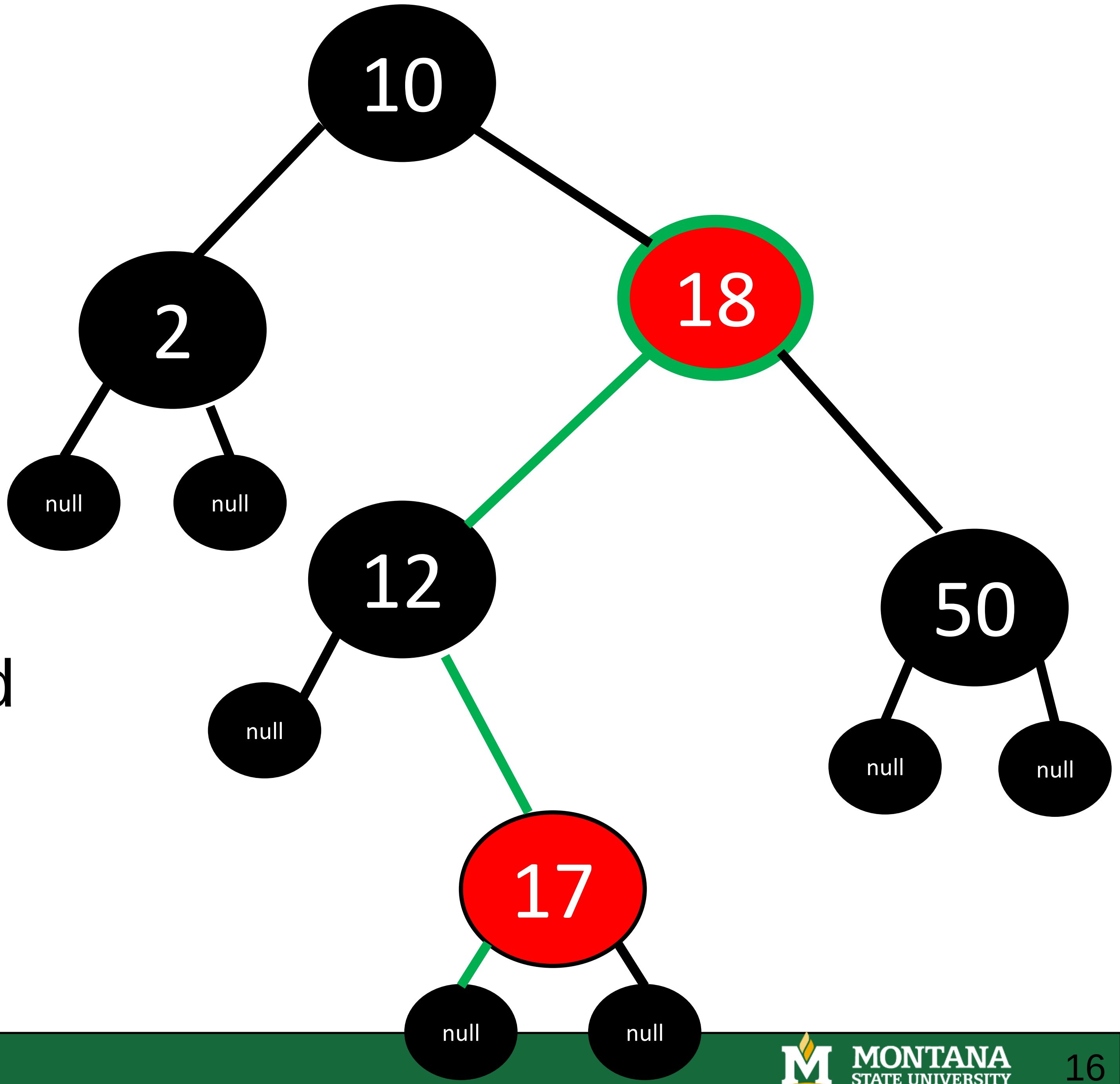
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes



Red-Black Tree Rules

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

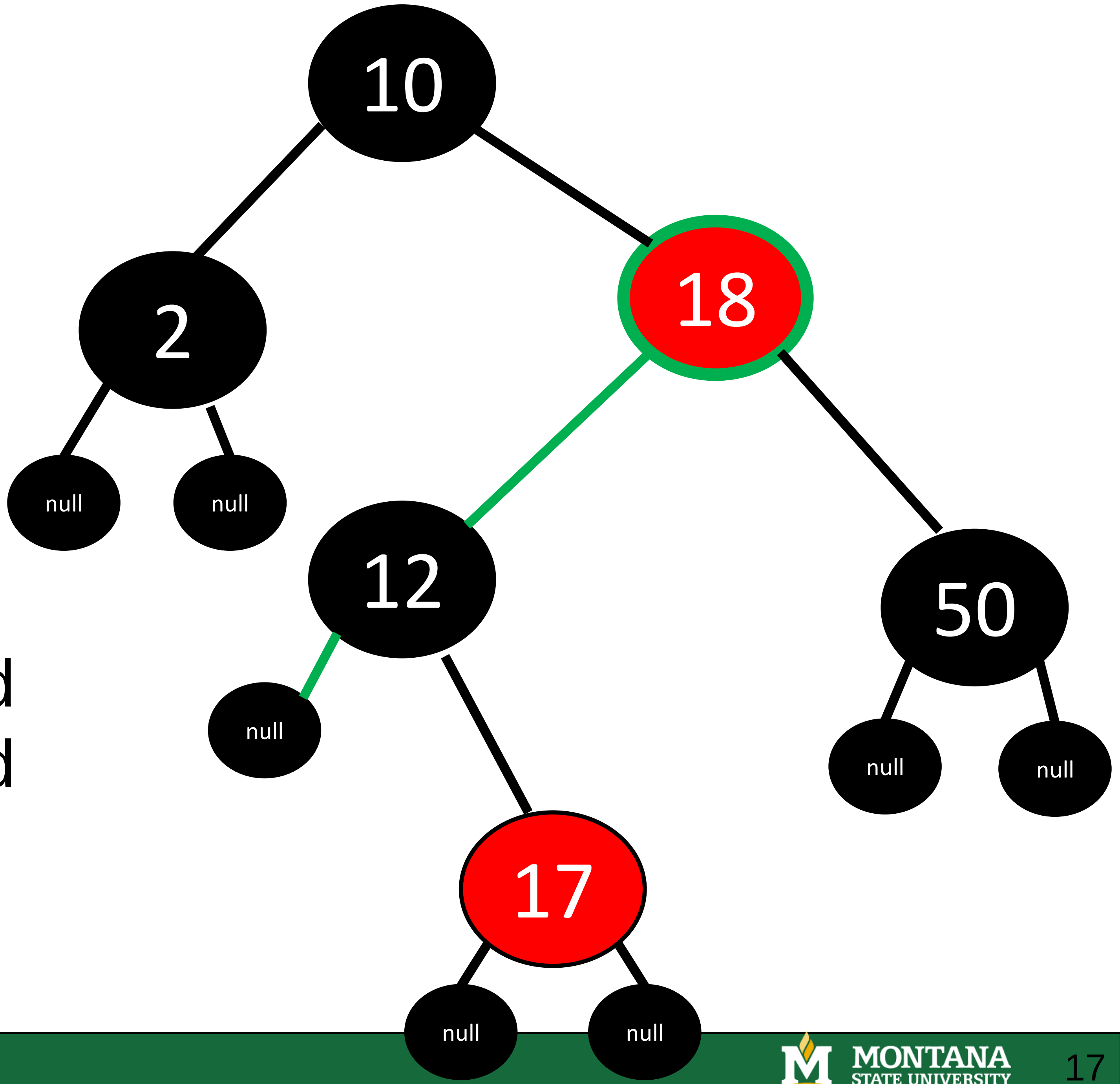
Path 1: 2 black nodes visited



Red-Black Tree Rules

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

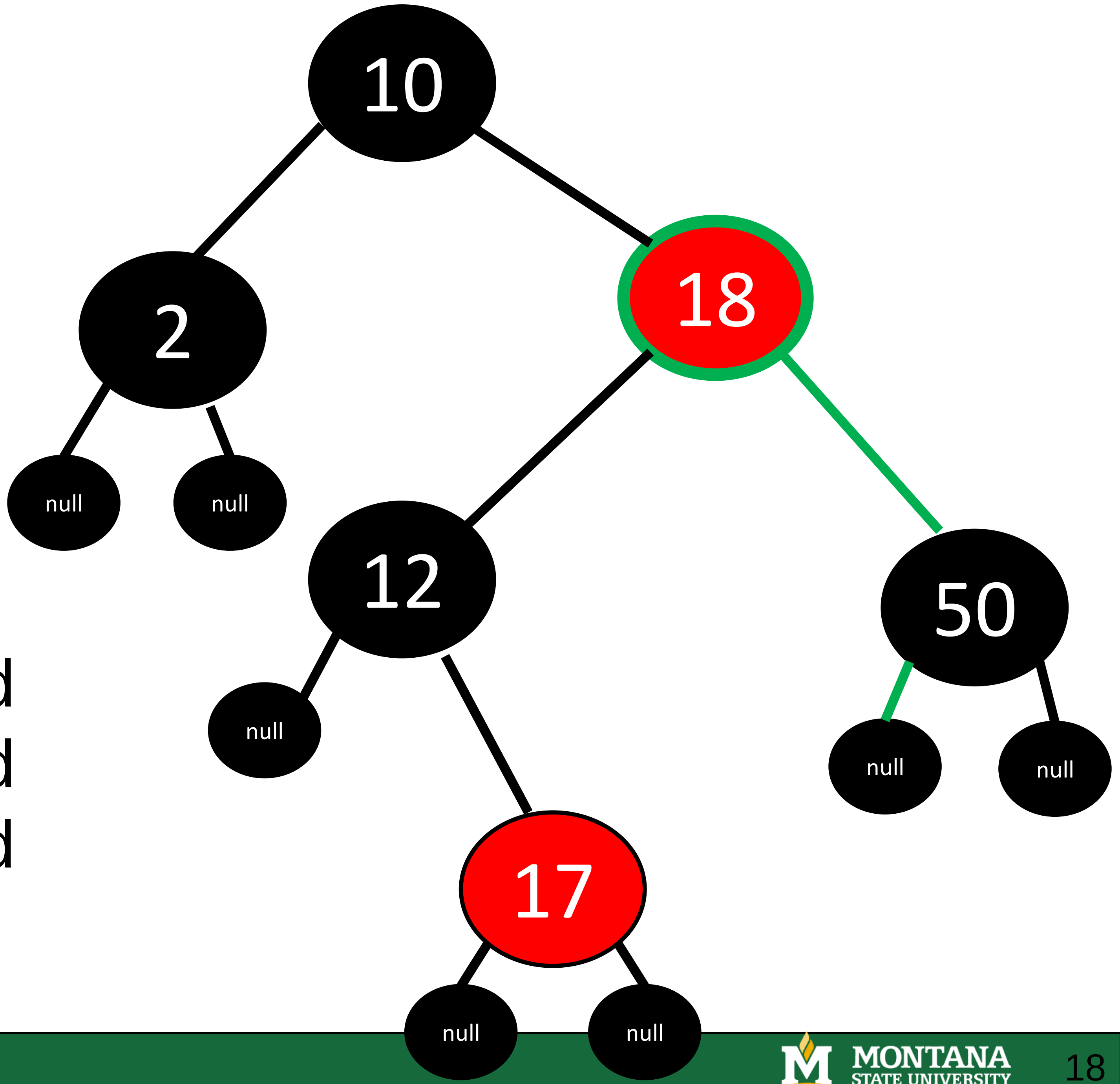
Path 1: 2 black nodes visited
Path 2: 2 black nodes visited



Red-Black Tree Rules

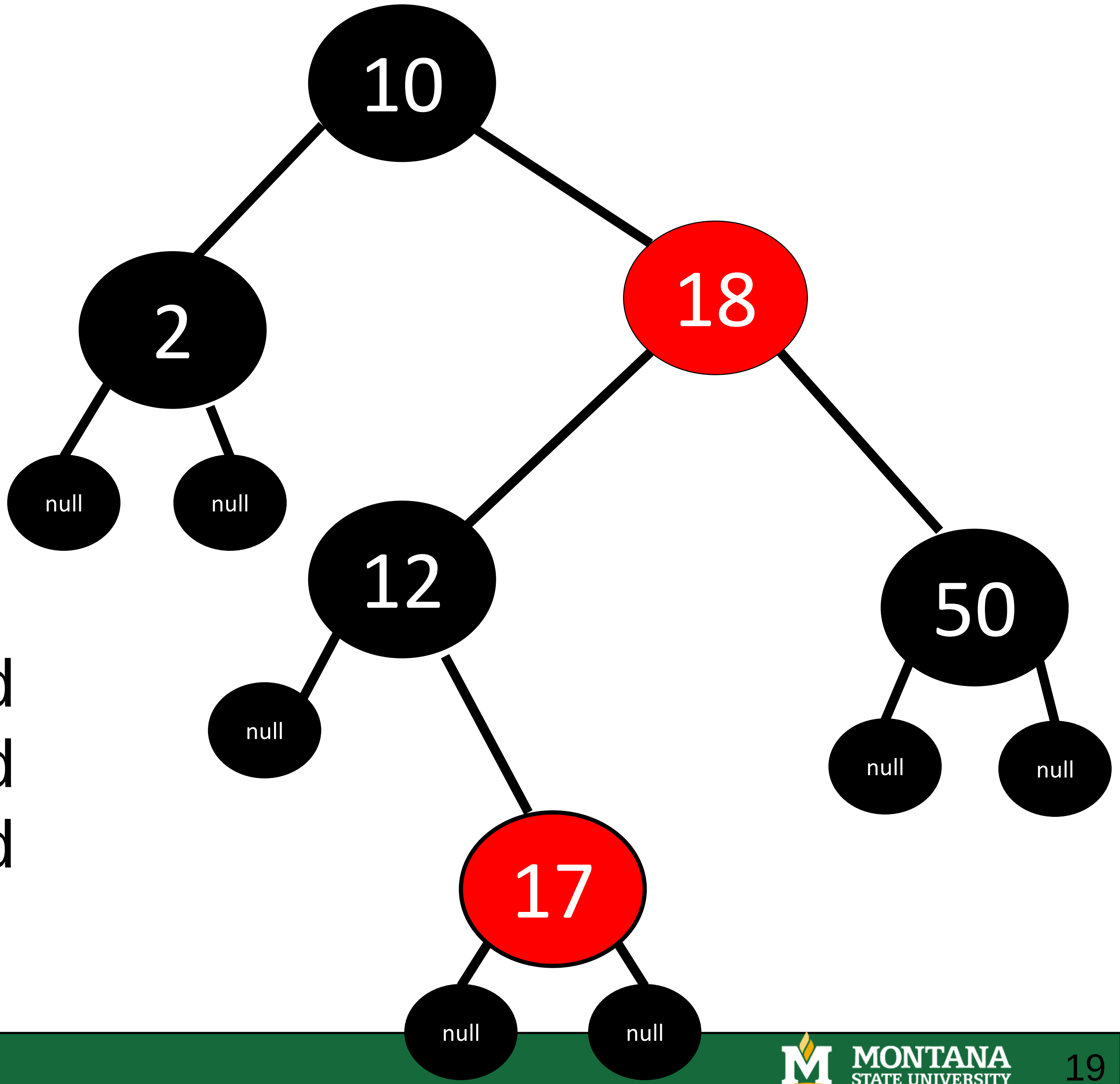
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

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

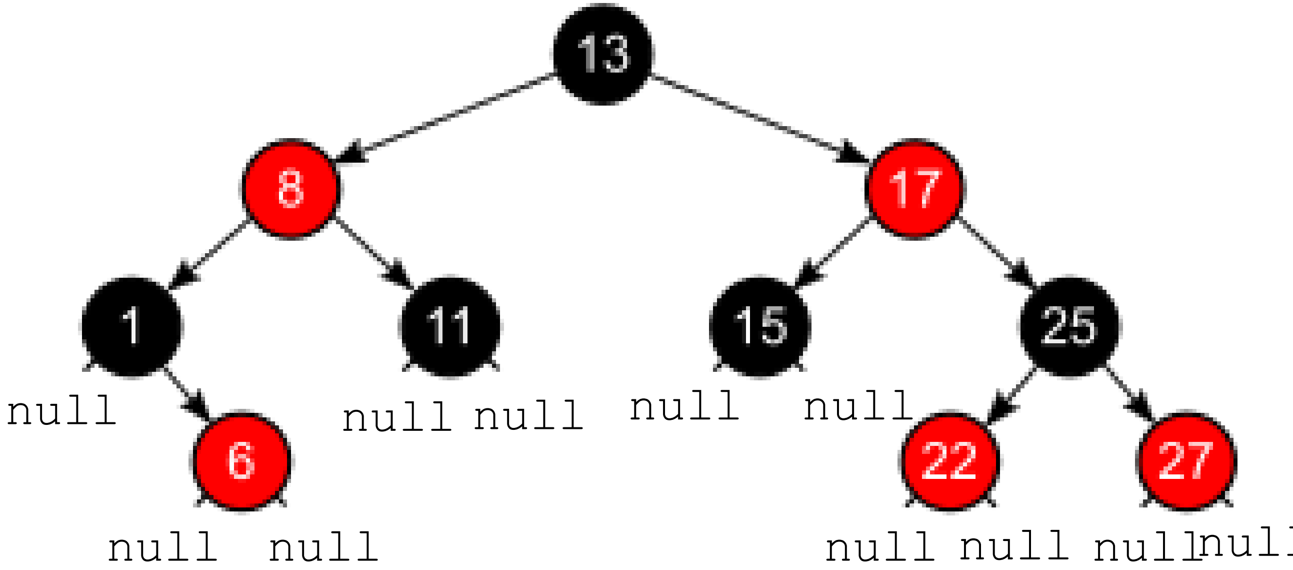


Path 1: 2 black nodes visited
Path 2: 2 black nodes visited
Path 3: 2 black nodes visited

Red-Black Tree Rules

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

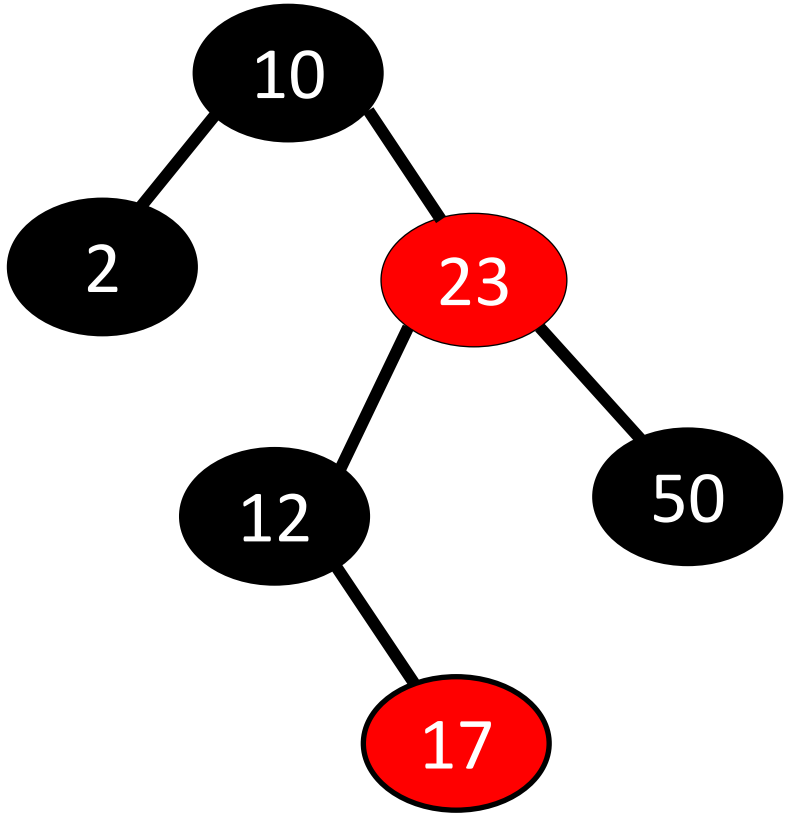
When we **insert** or **delete** something from a Red-Black tree, the new tree may **violate** one of these rules



Red-Black Tree Insertion/Deletion

`insert(15)`

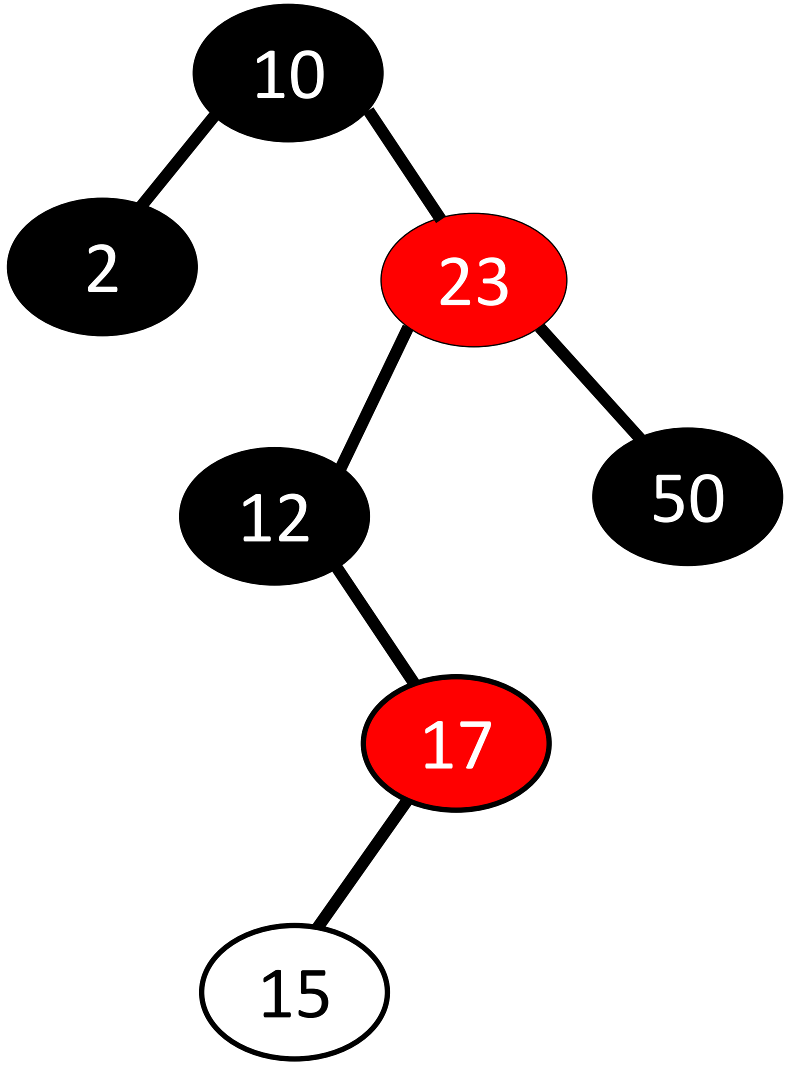
Step 1: Do the normal BST insertion



Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion

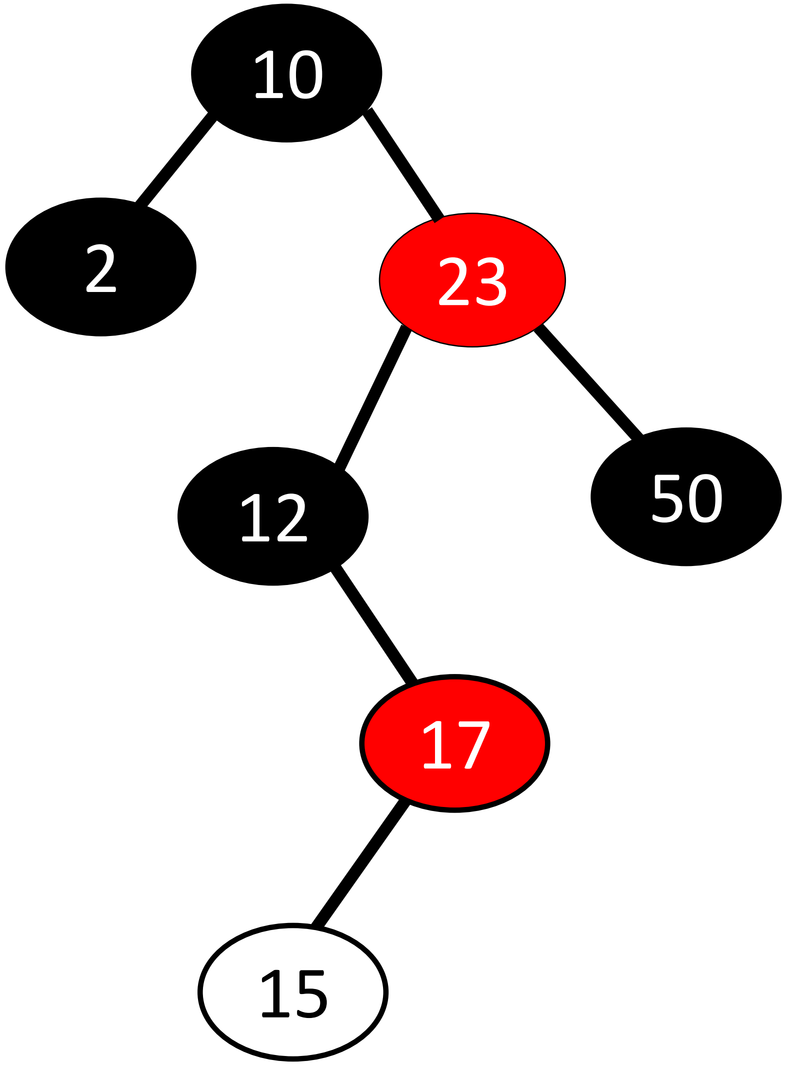


Our tree no longer has log(n) height, so we need to do some operations to reduce the height of the tree

Red-Black Tree Insertion/Deletion

`insert(15)`

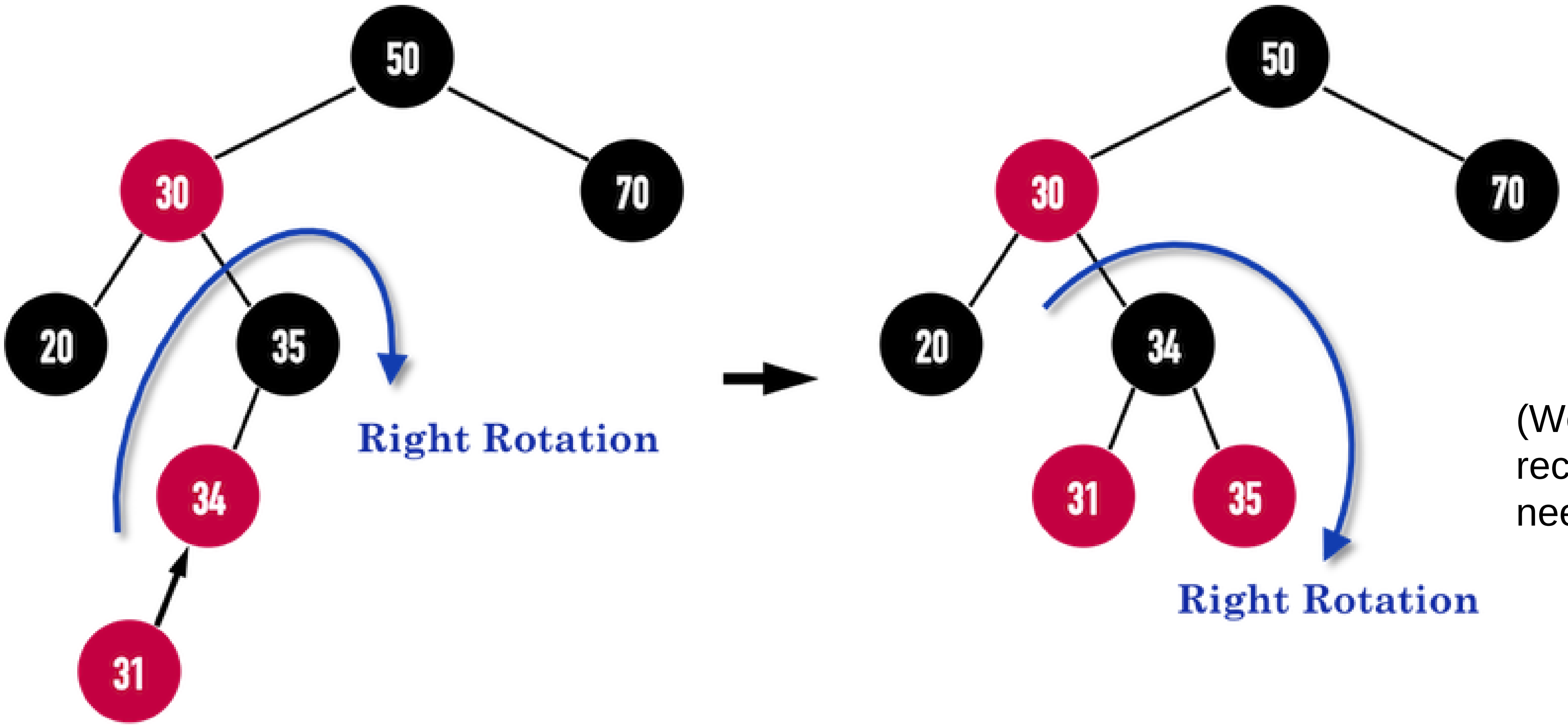
Step 1: Do the normal BST insertion



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**

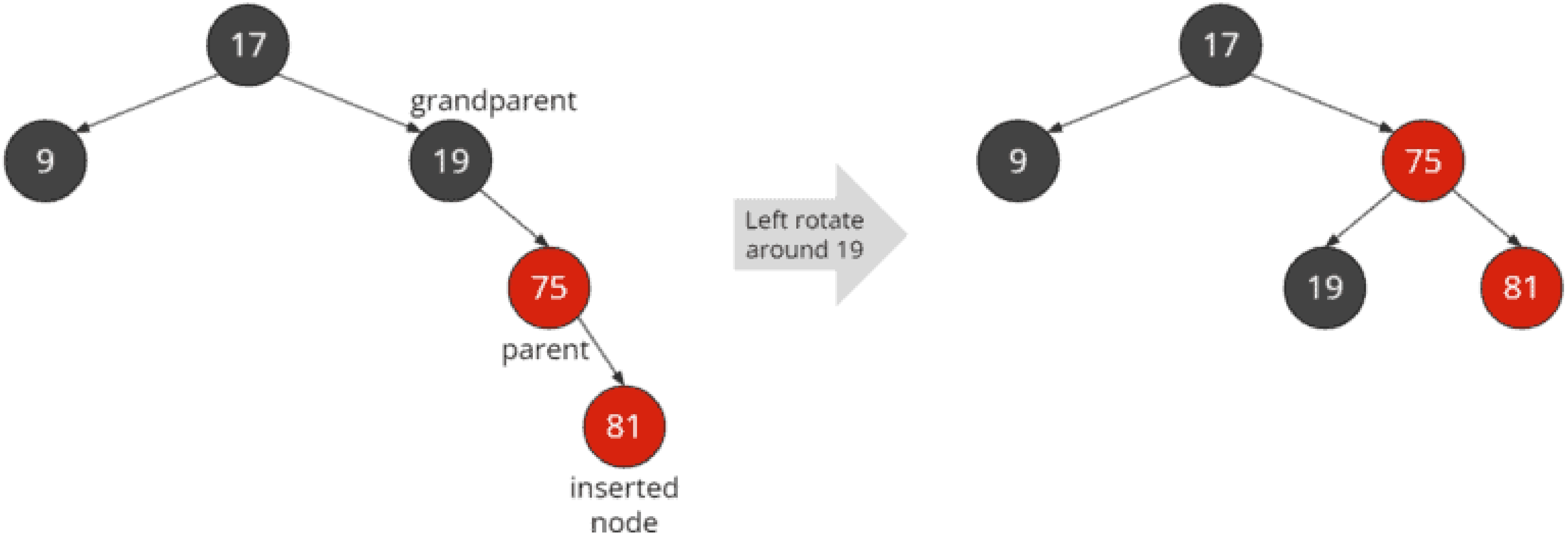
Red-Black Tree Rotation



(We also do some recoloring if needed!)

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

Red-Black Tree Rotation

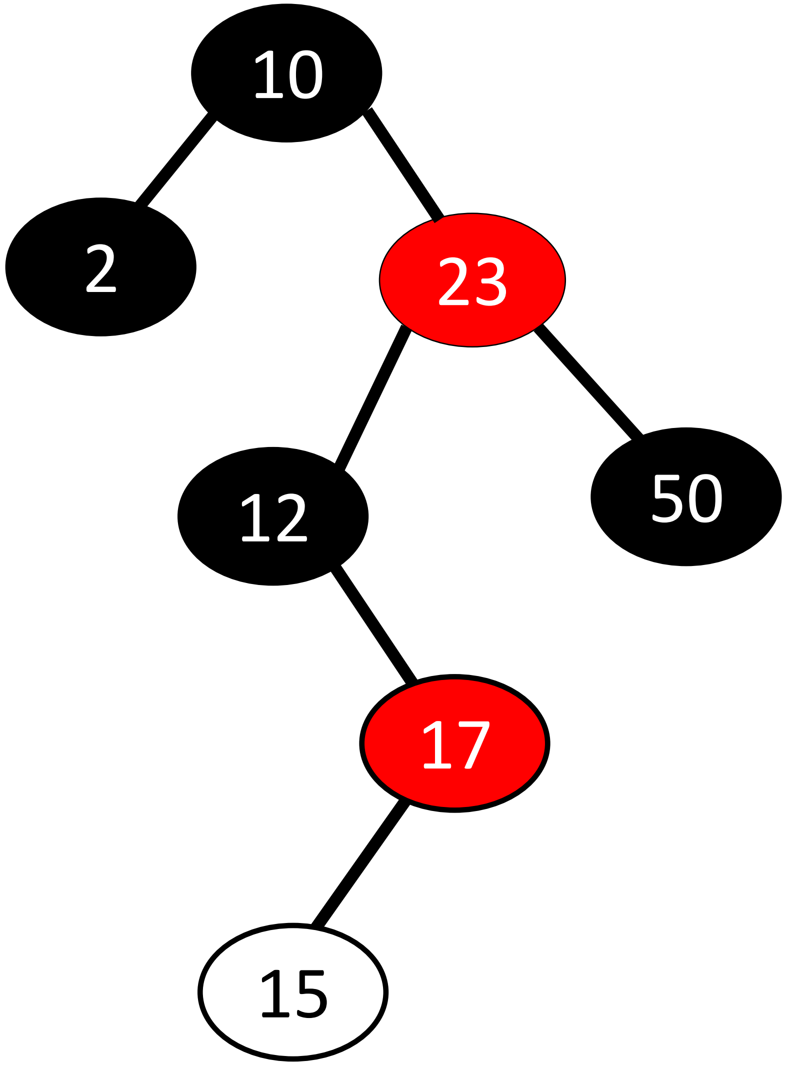


Local transformation (we rotate just a section– not the entire tree)

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion



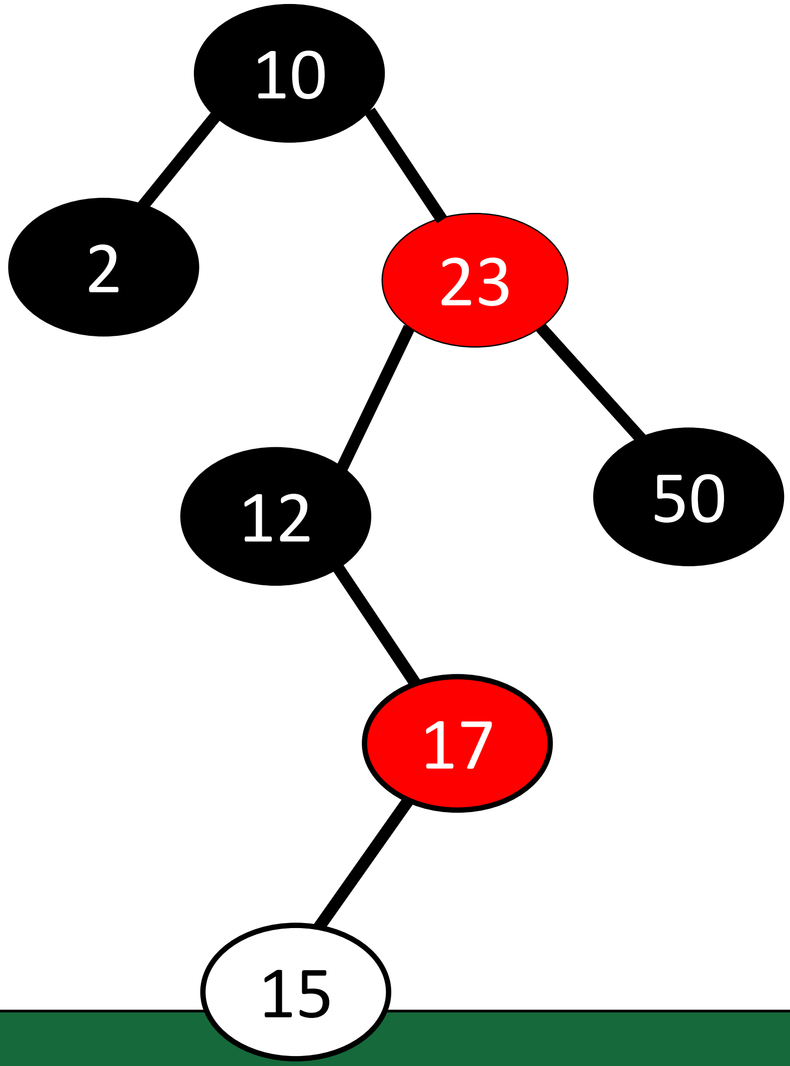
Our tree no longer has log(n) height, so we need to do some operations to reduce the height of the tree

These operations are known as **rotations**

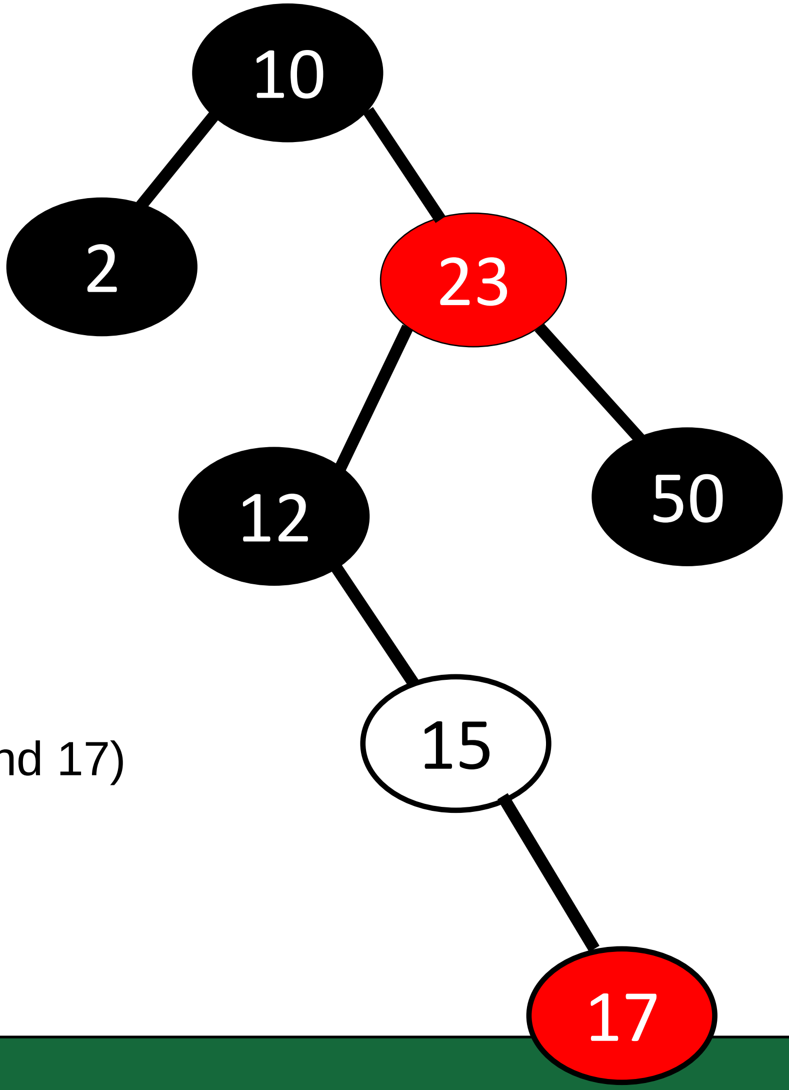
Red-Black Tree Insertion/Deletion

`insert(15)`

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



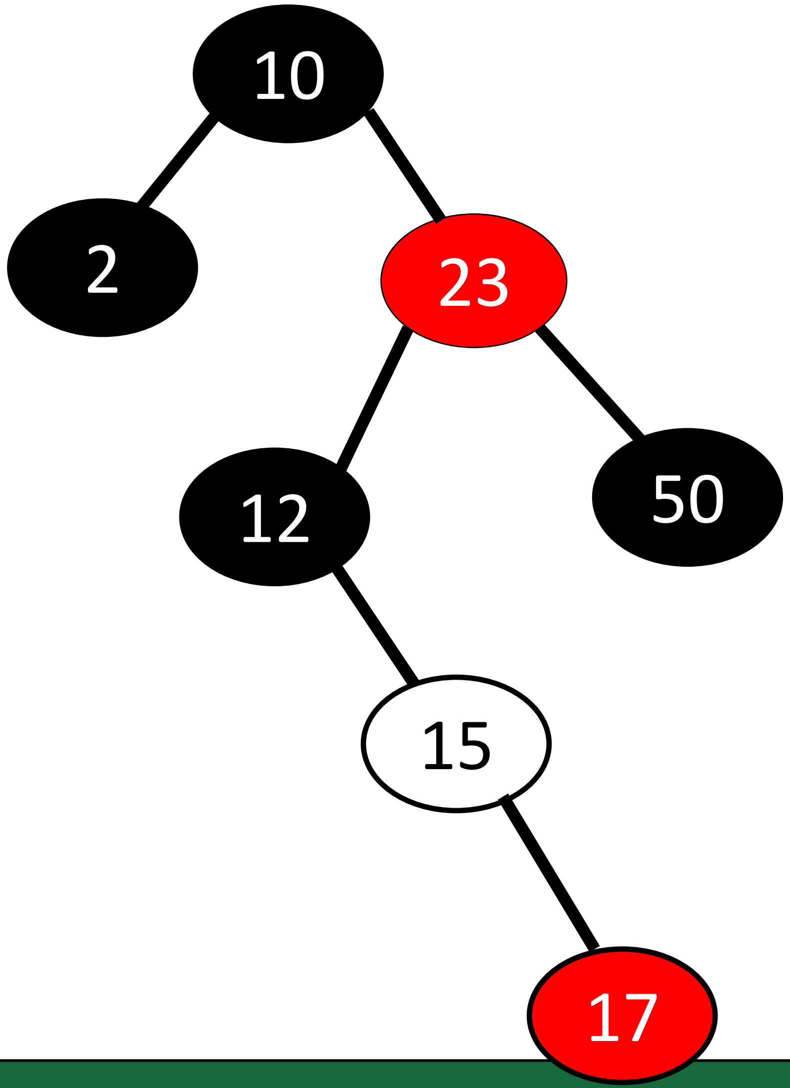
(Rotate Right around 17)



Red-Black Tree Insertion/Deletion

`insert(15)`

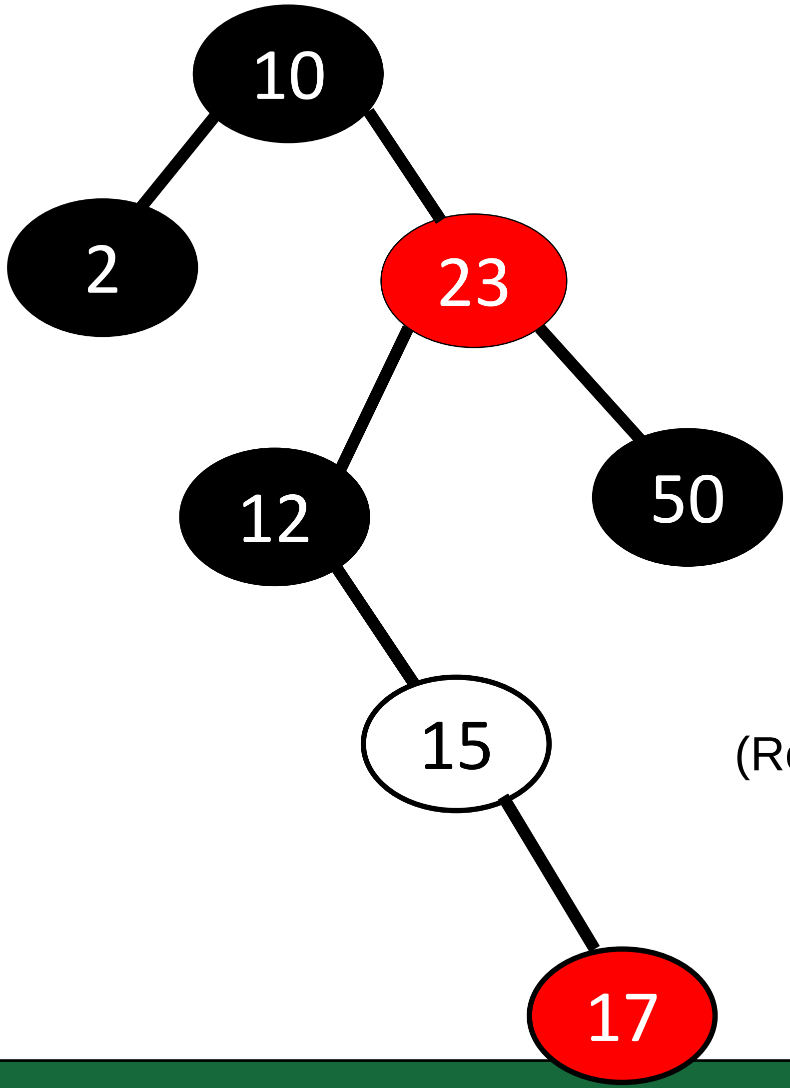
Step 1: Do the normal BST insertion
Step 2: Do rotation(s)



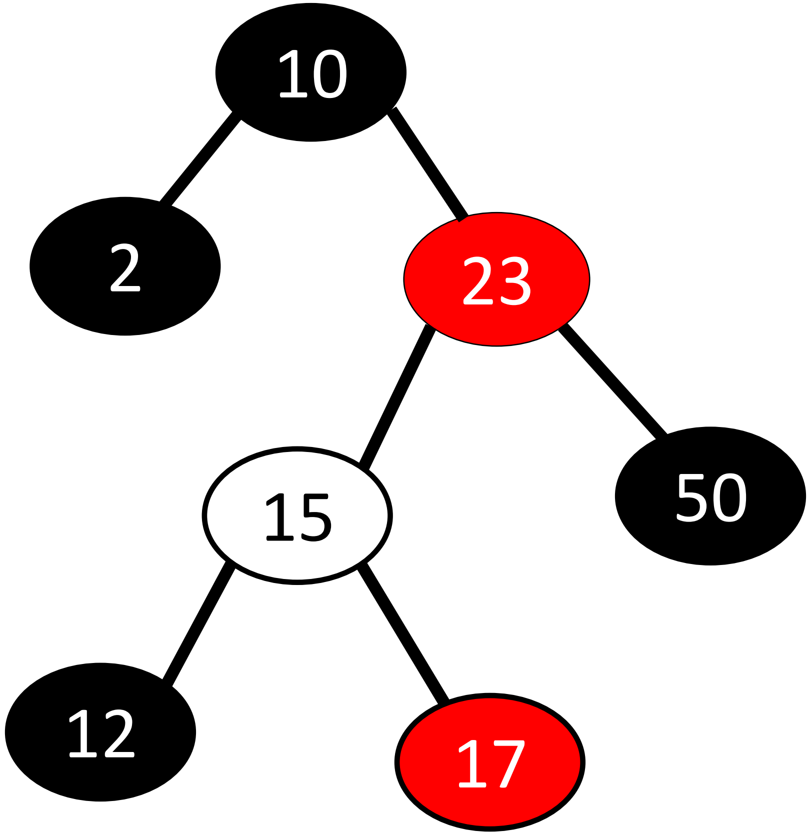
Red-Black Tree Insertion/Deletion

`insert(15)`

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



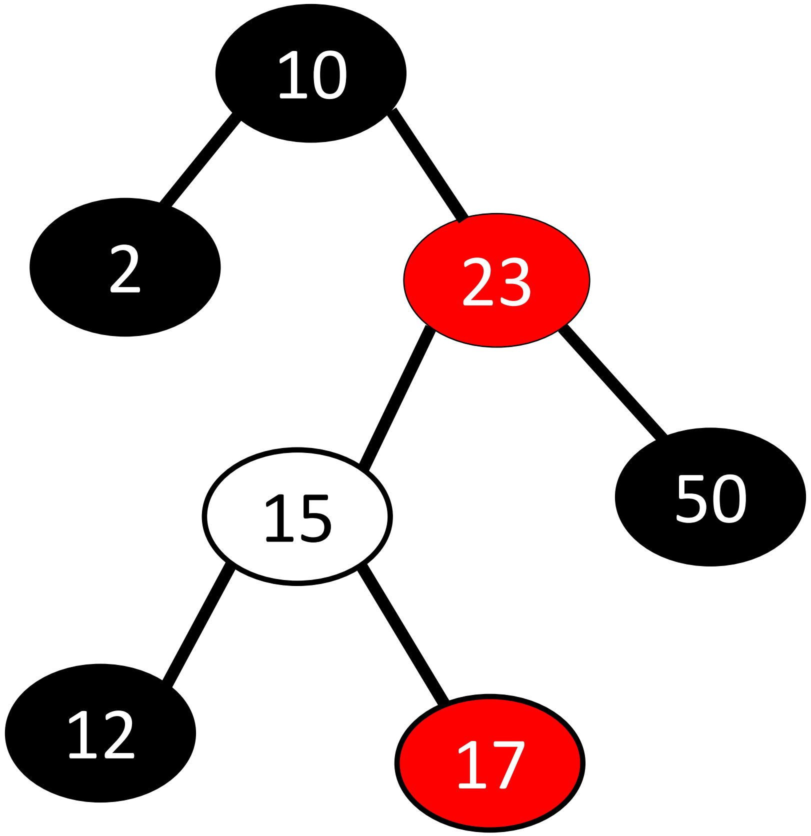
(Rotate left around 12)



Red-Black Tree Insertion/Deletion

`insert(15)`

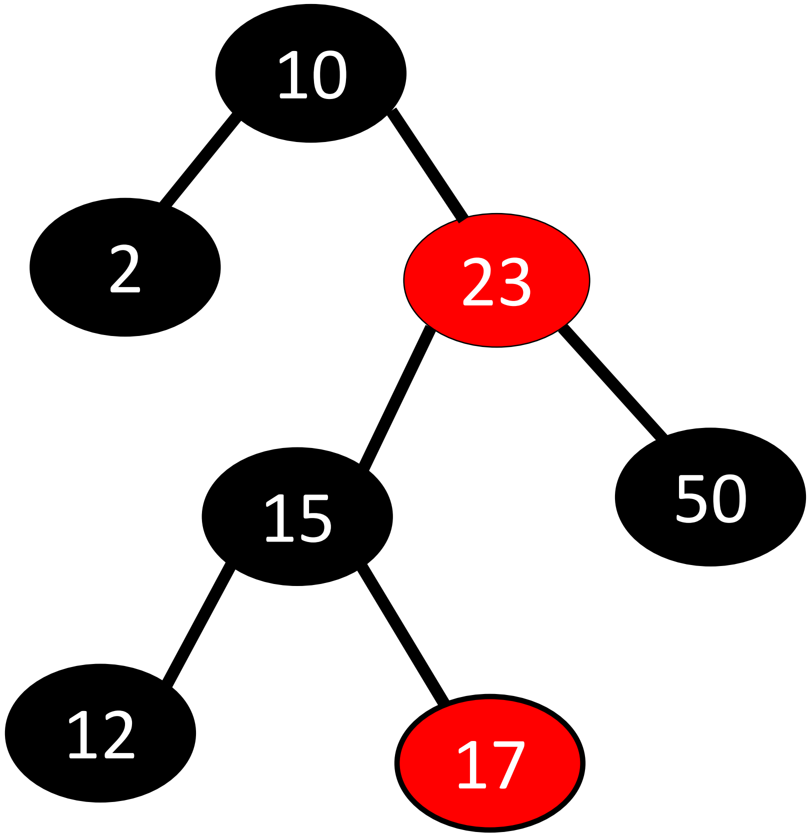
Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor



Red-Black Tree Insertion/Deletion

`insert(15)`

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

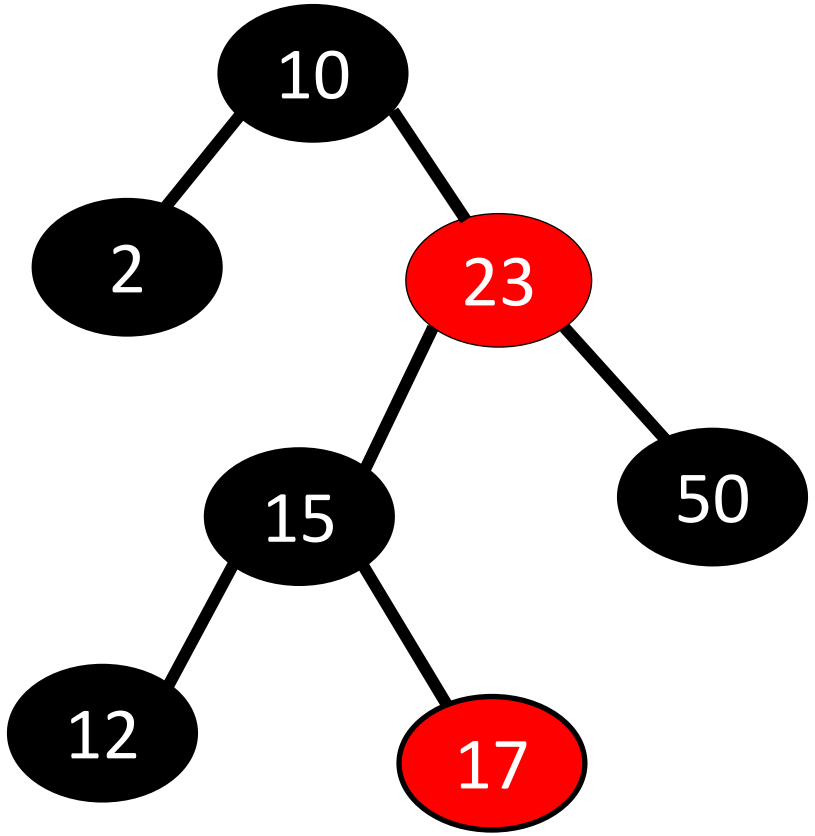


15 has to be black because….

Red-Black Tree Insertion/Deletion

`insert(15)`

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



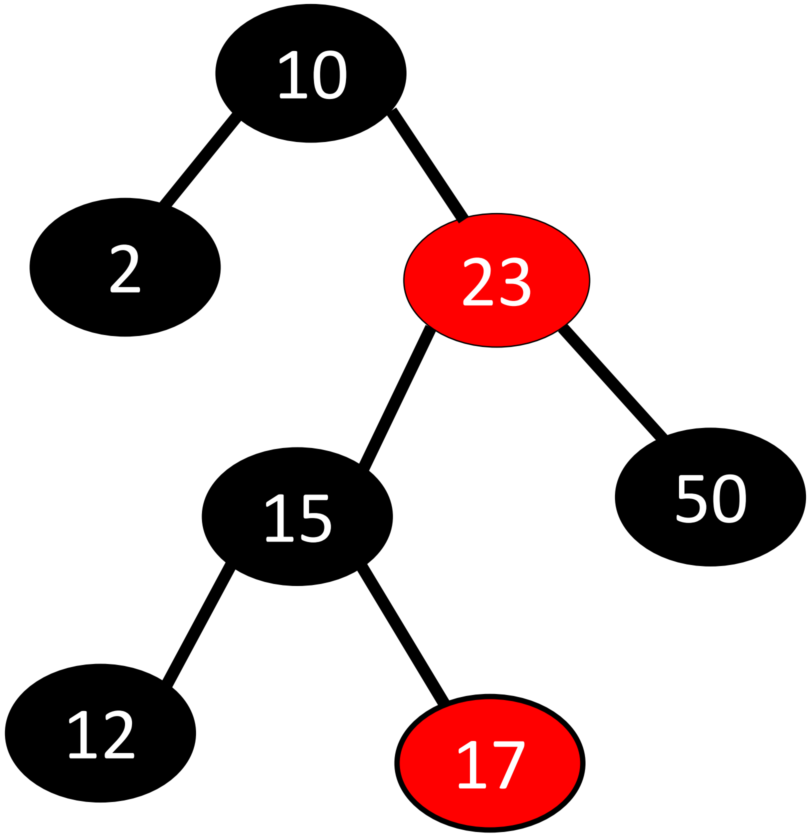
3. If a node is **red**, both children must be **black**

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

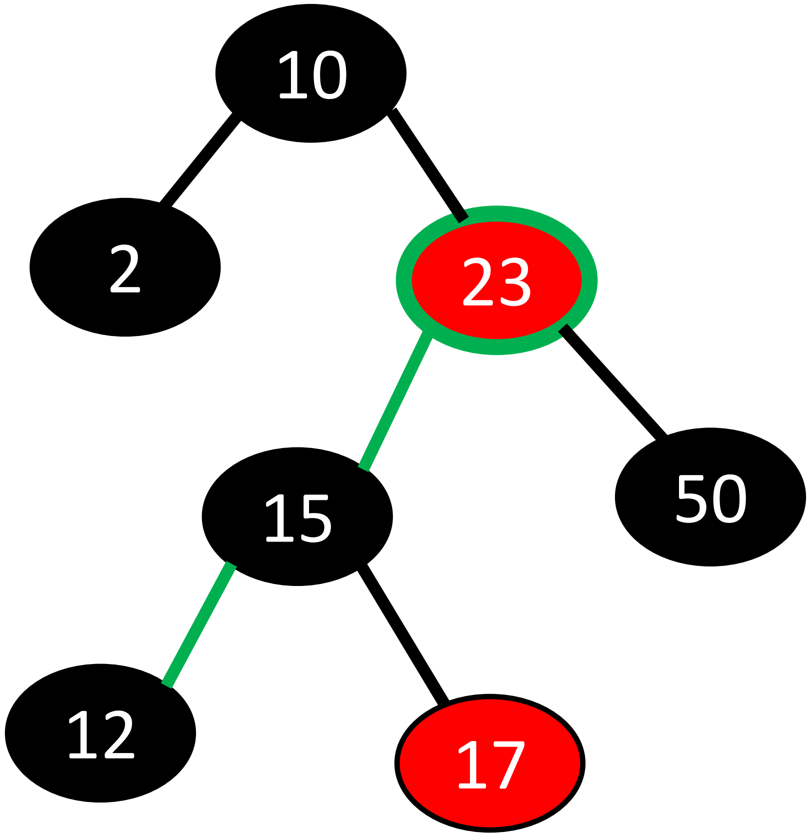


Is this a Red-Black tree?

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?

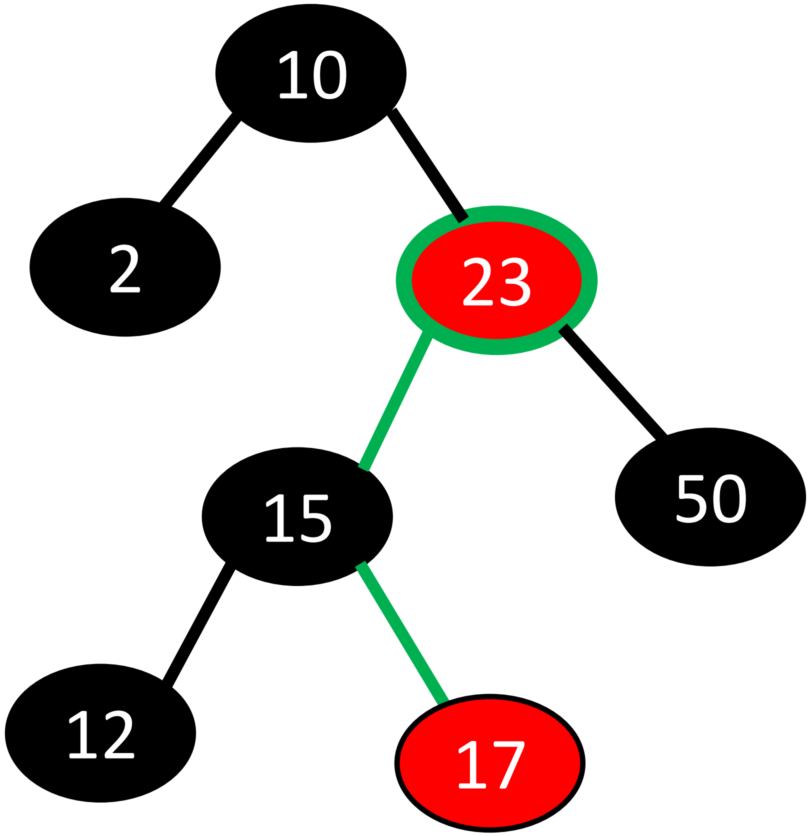
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

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?

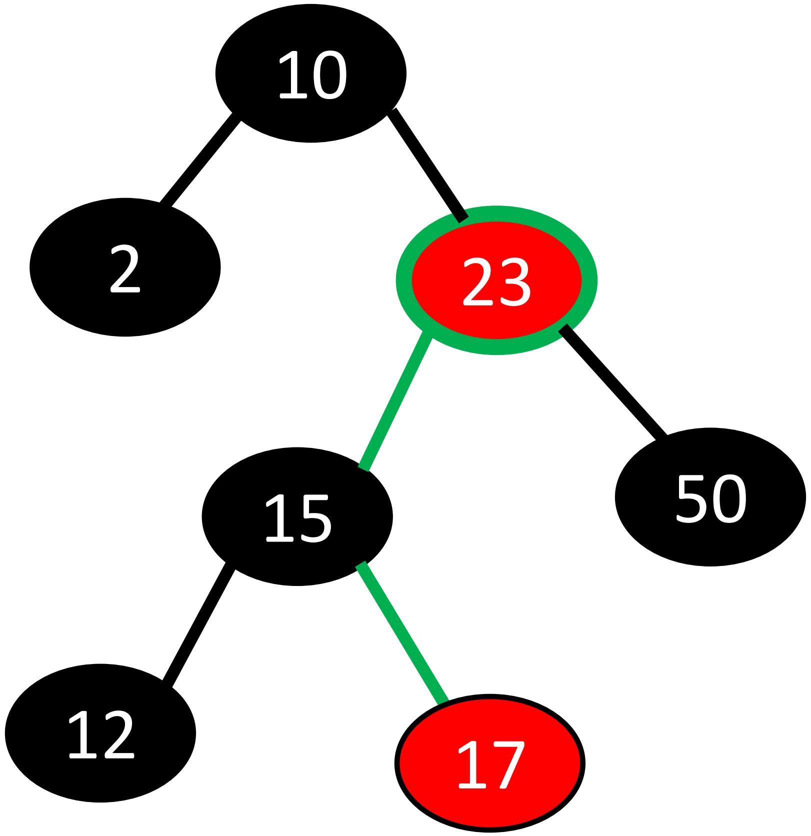
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

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

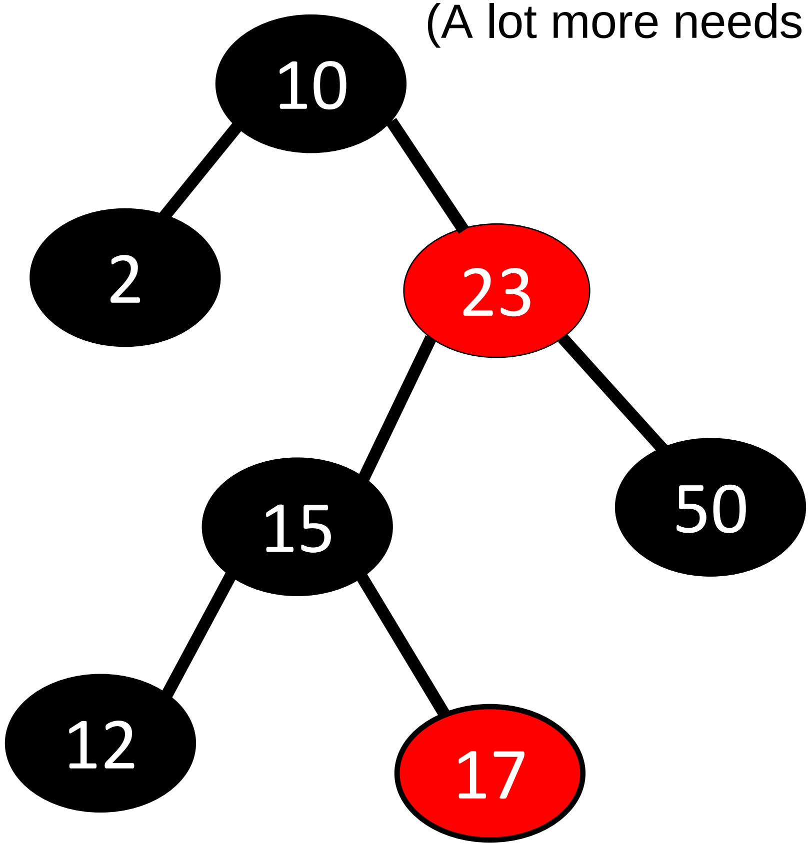
Path 1: 3 black nodes (including null nodes)
Path 2: 2 black nodes (including null nodes)

Red-Black Tree Insertion/Deletion

`insert(15)`

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

(A lot more needs to be done here)

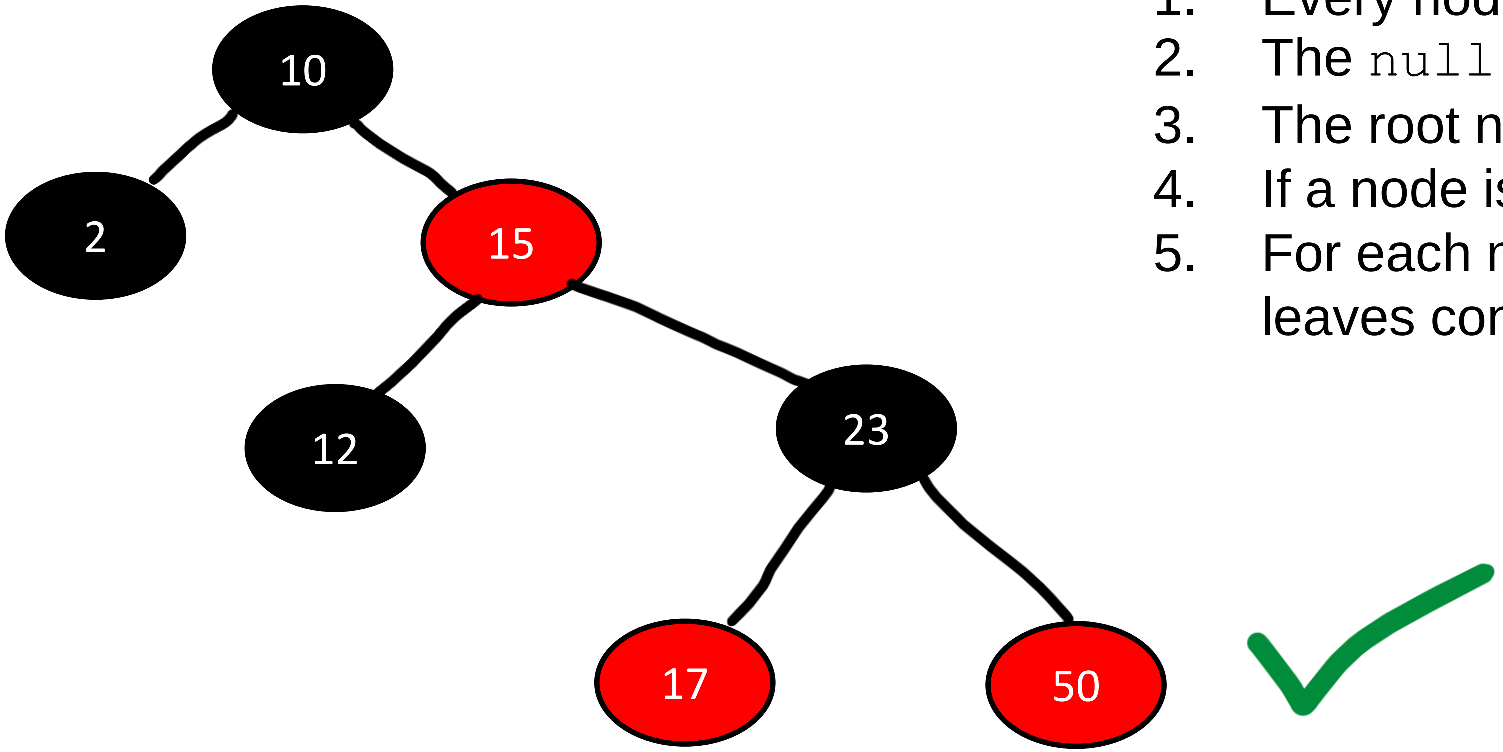


1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**
4. If a node is **red**, both children must be **black**
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

Red-Black Tree Insertion/Deletion

`insert(15)`

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

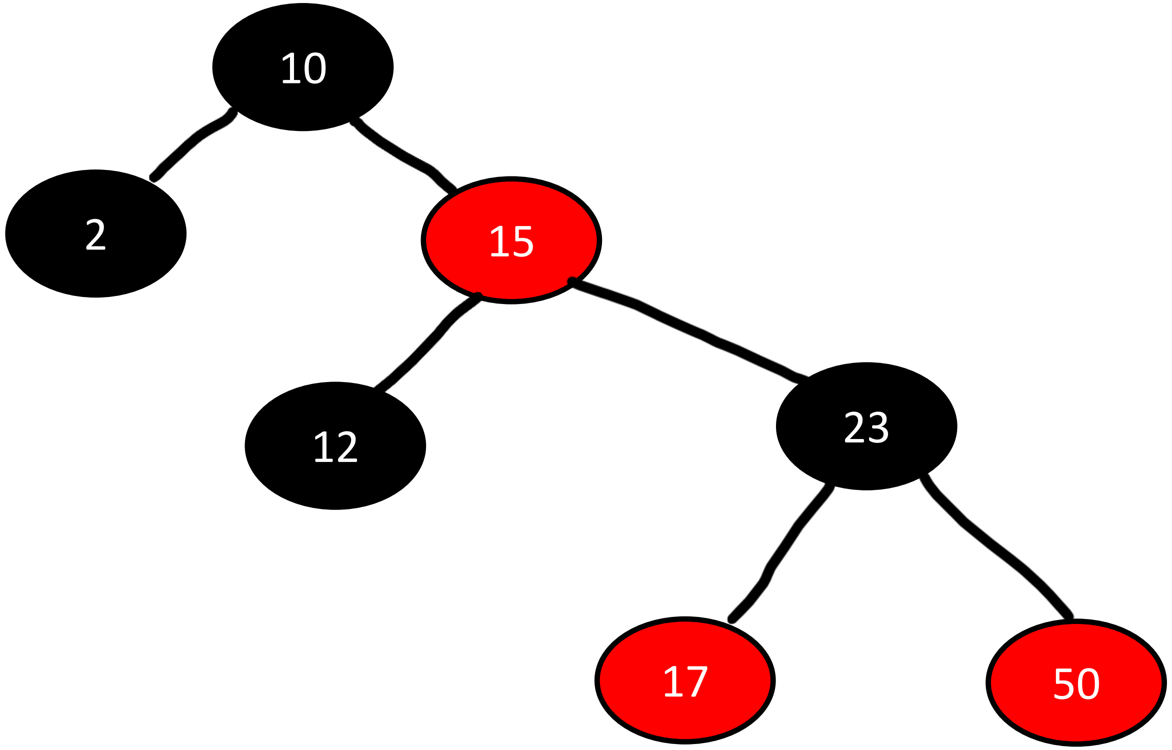


1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**
4. If a node is **red**, both children must be **black**
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

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

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)
```

(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

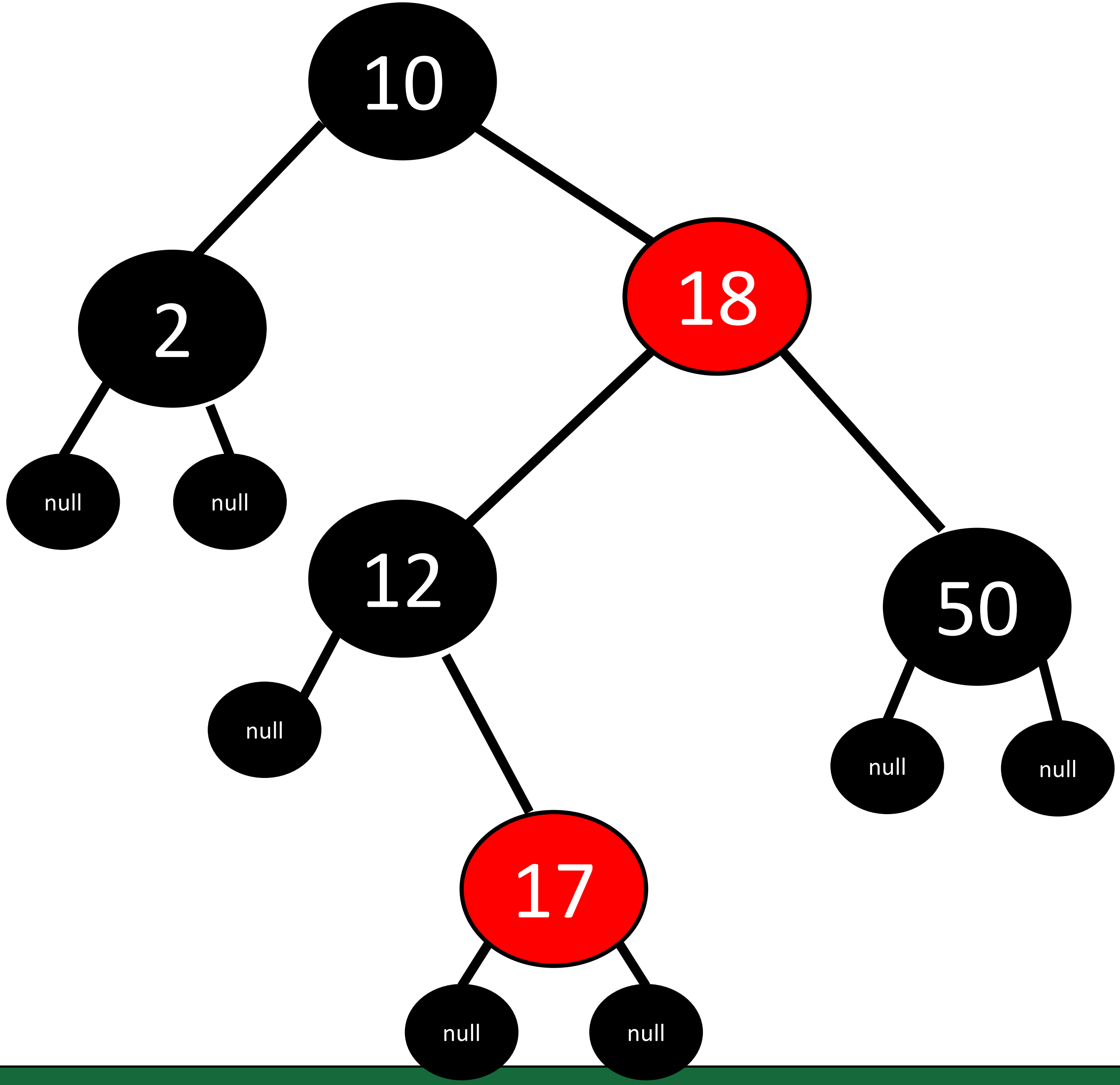
Guaranteed Balanced BST =

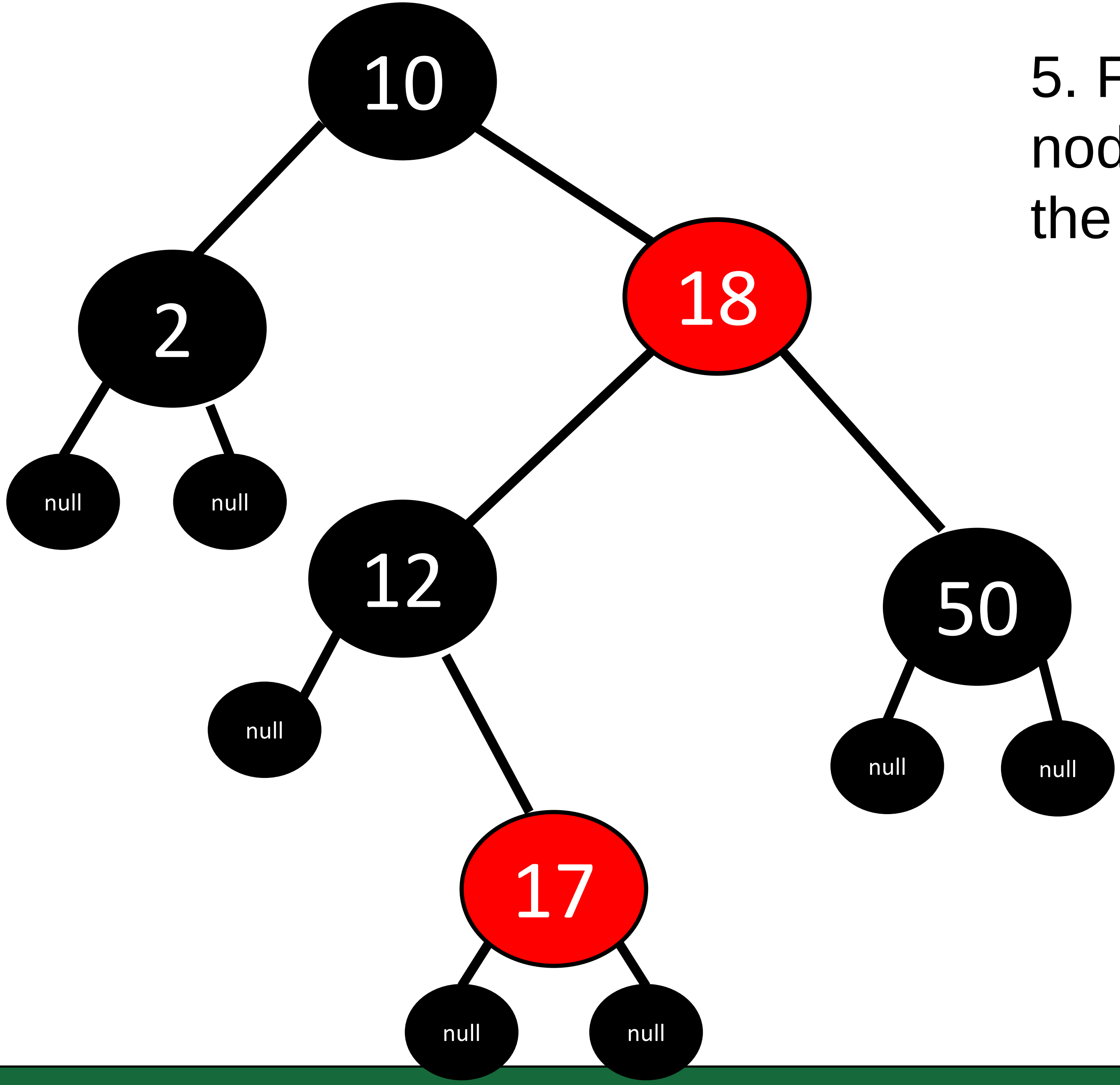
- ❑ O(logn) insertion
- ❑ O(logn) deletion
- ❑ O(logn) Searching/Contains

Adding Red/Black functionality to a BST does not affect the running time

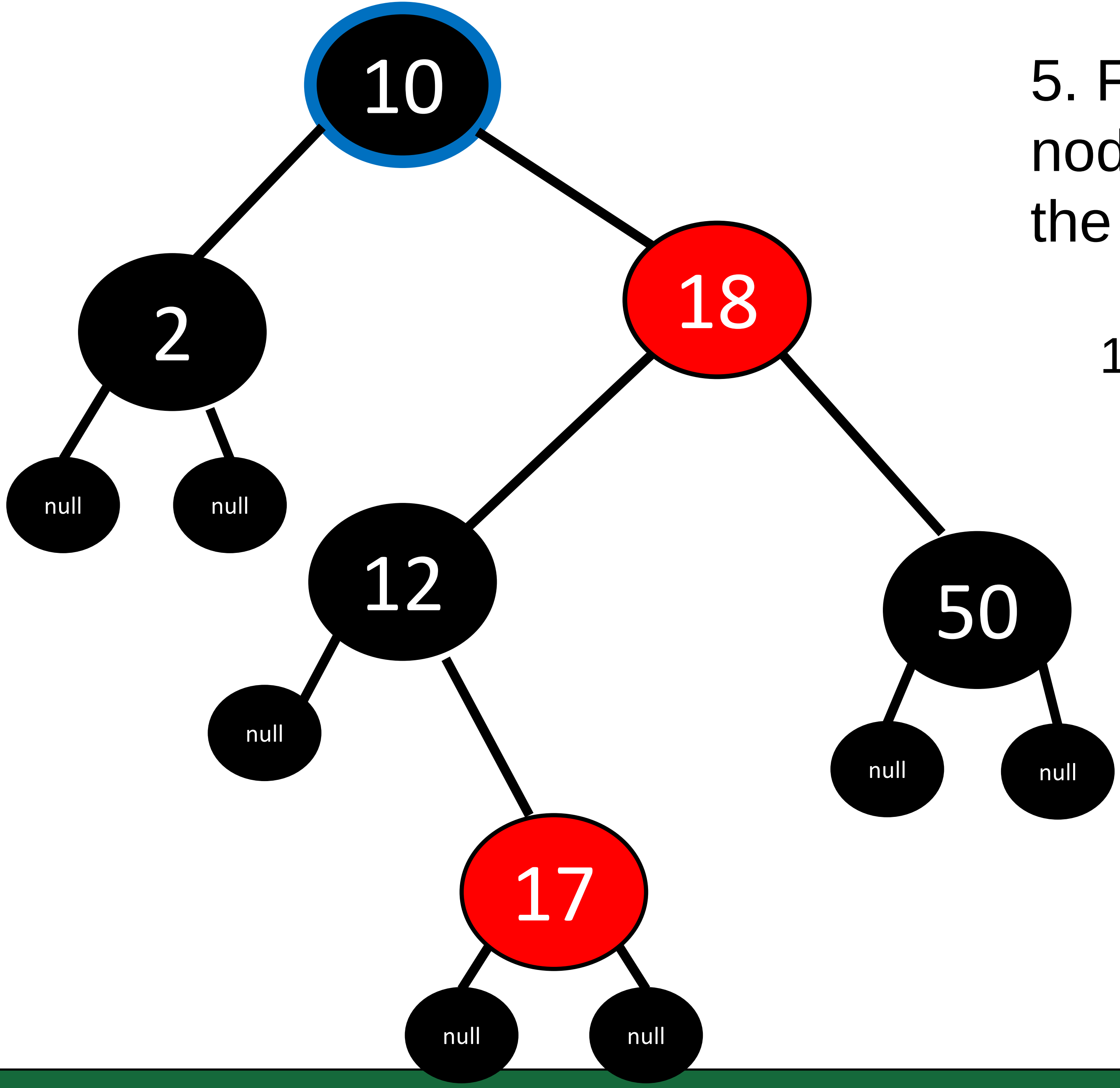
There are also BSTs called **AVL tree** and **2-3 trees** that serve the same purpose of RB trees

You will never have to write code for a red black tree, but you should know the purpose of red black trees, and be able to verify if a red black tree is valid or not


10
2
18
null
null
12
50
null
null
null
17
null
null

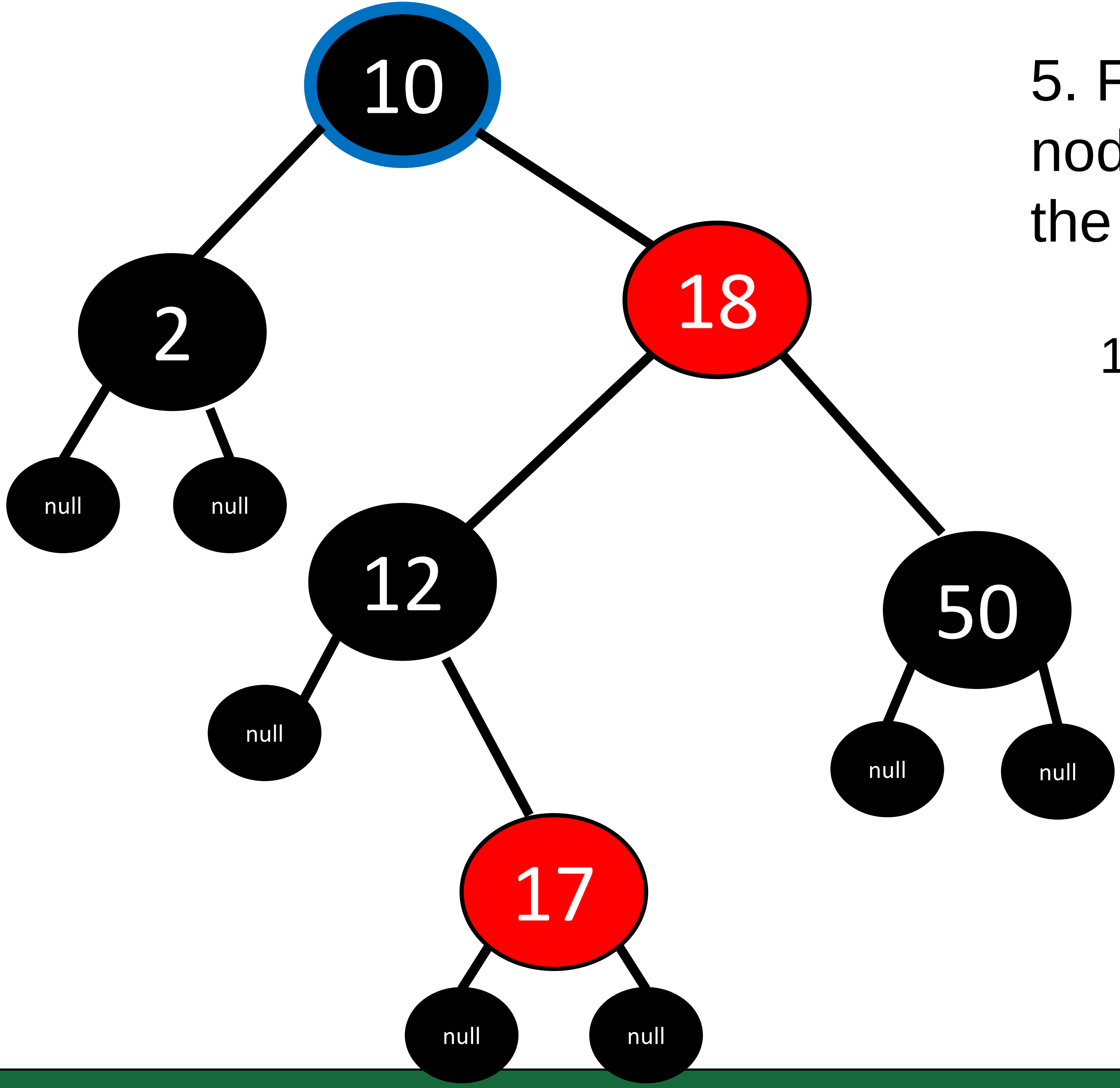


5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes



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

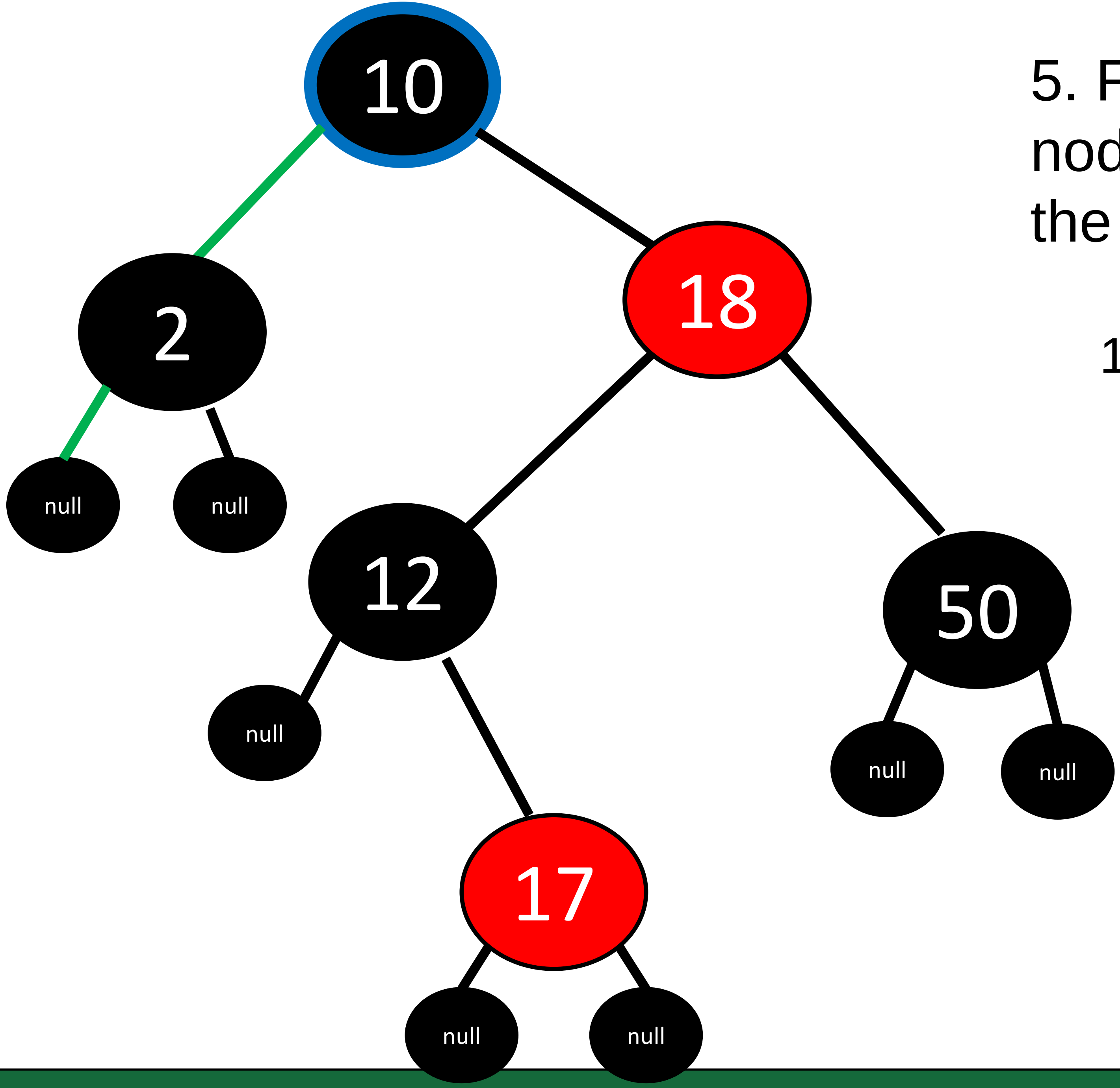
1. Get Leaf Nodes from starting node



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

1. Get Leaf Nodes from starting node

```
leaves = [ 2, 17, 50 ]
```



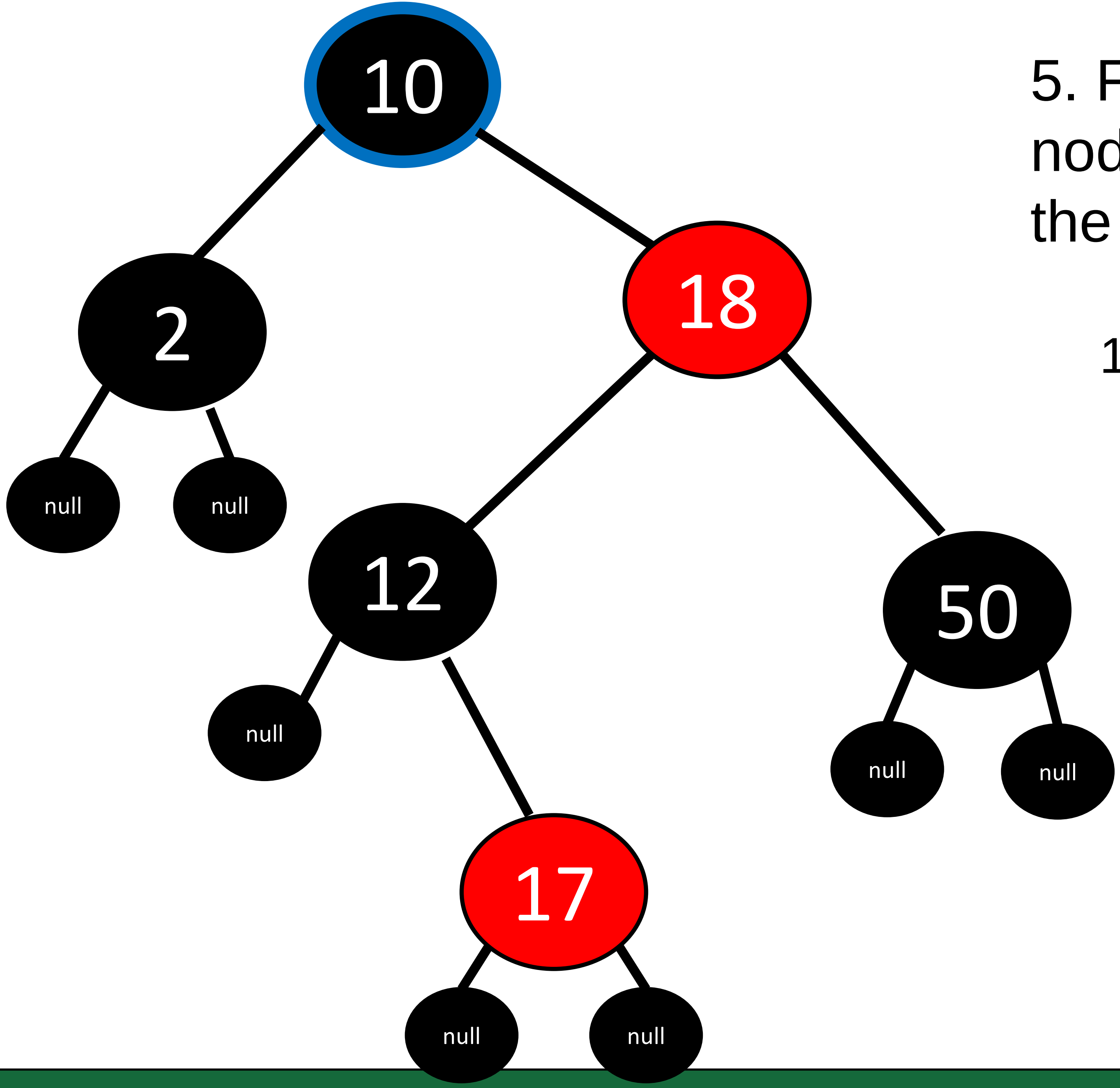
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

1. Get Leaf Nodes from starting node

```
leaves = [ 2, 17, 50 ]
```

2. Calculate the path from leaf to root, and count the number of black nodes visited

```
2 :  3
17:  3
50:  3
```



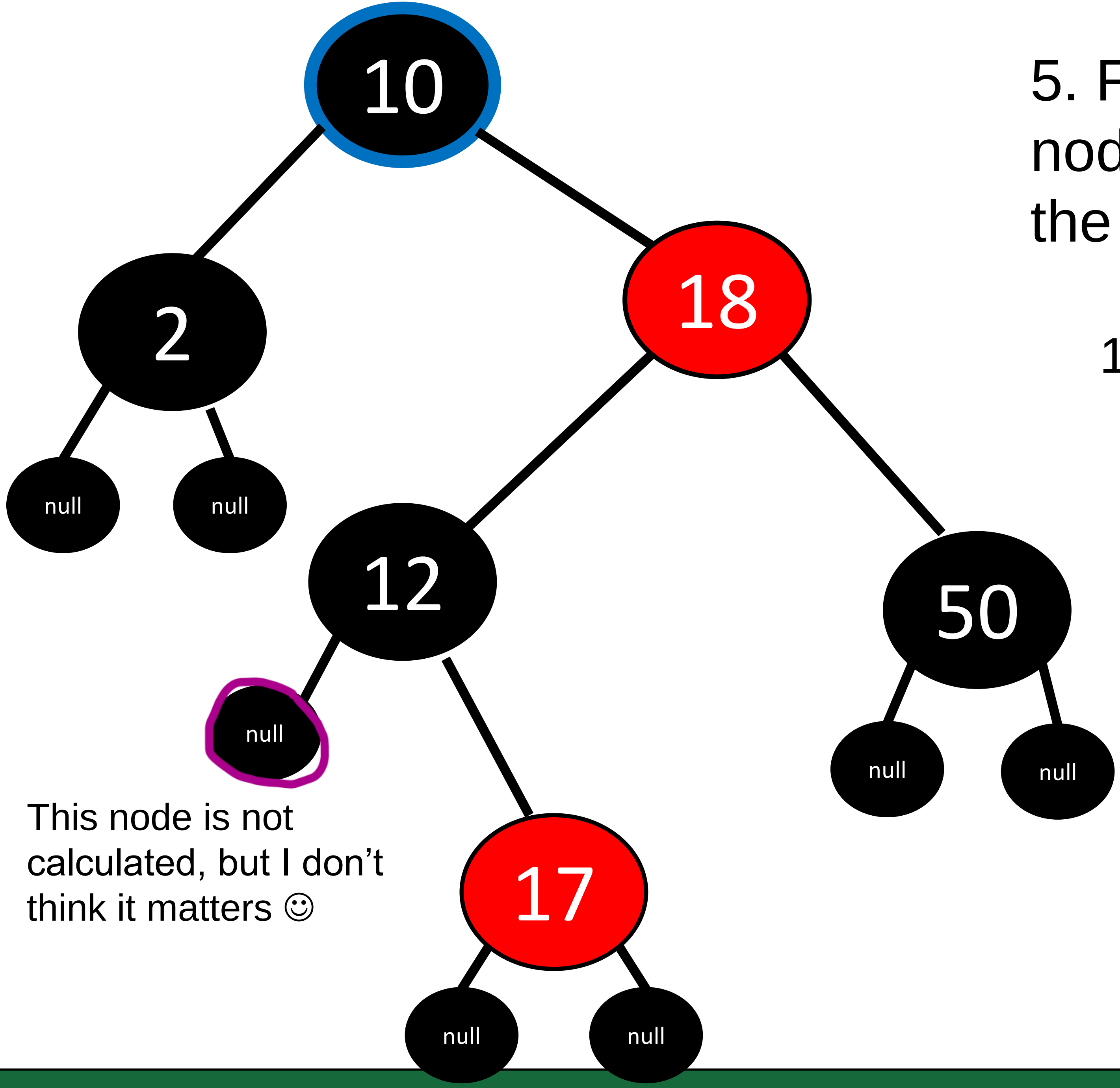
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

1. Get Leaf Nodes from starting node

```
leaves = [ 2, 17, 50 ]
```

2. Calculate the path from leaf to root, and count the number of black nodes visited

```
2 :  3
17:  3
50:  3
```



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

1. Get Leaf Nodes from starting node

```
leaves = [ 2, 17, 50 ]
```

2. Calculate the path from leaf to root, and count the number of black nodes visited

```
2 : 3
17: 3
50: 3
```

3. Make sure all these numbers are the same