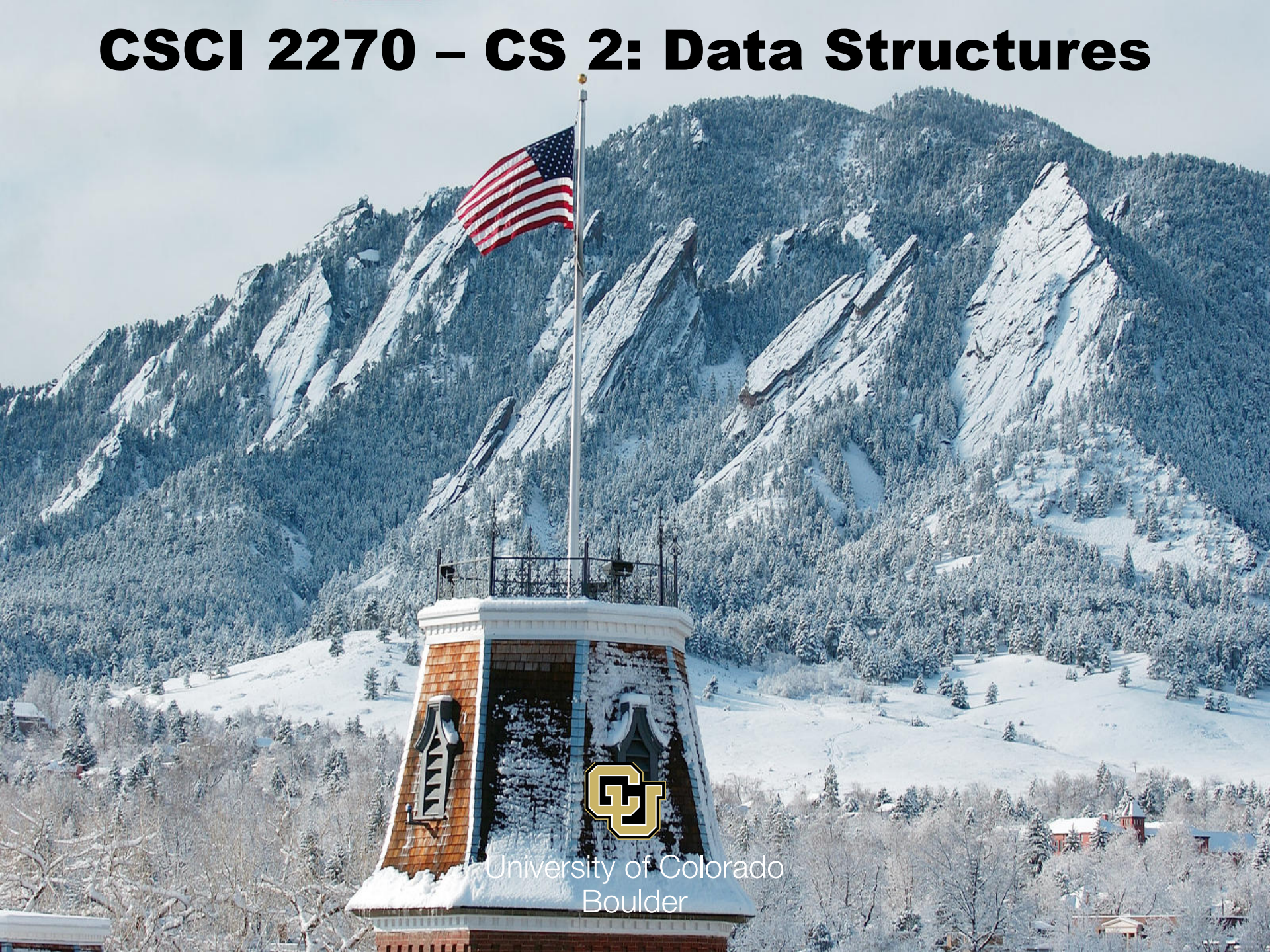
CSCI 2270 – CS 2: Data Structures



University of Colorado
Boulder

Reminders

Topics

- Red-Black Trees

Red-Black Trees – Why?

- Most BST operations take O(h) time
- However, tress can become skewed and turn into O(n)
- So, RBT's guarantee O(log n)!

Red-Black Trees

- A red-black tree is a binary search tree
- Each node in the BST is assigned a color, either red or black
- Each node in a red-black tree has at least the following properties:
  - color
  - key
  - left child
  - right child
  - parent (except root node)

Red-Black Tree Properties

- **Property 1:** A node is either red or black.
- **Property 2:** The root node is black.
- **Property 3:** Every leaf (NULL) node is black.
- **Property 4:** If a node is red, then both of its children must be black.
- **Property 5:** For each node in the tree, all paths from that node to the leaf nodes contain the same number of black nodes.

Red-Black Tree Properties

- Another difference between a regular BST and a red-black tree is how the leaf nodes are represented. In a red-black tree, the leaf nodes are external sentinel nodes with all of the same properties as a regular node, but they are effectively empty nodes.

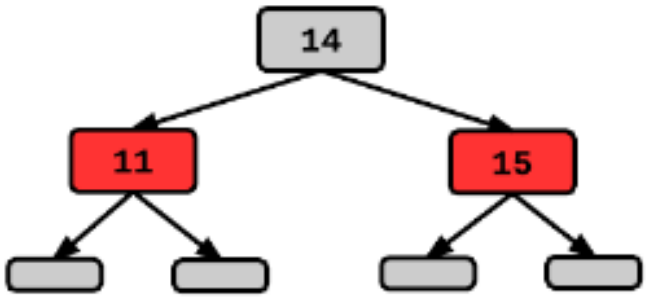
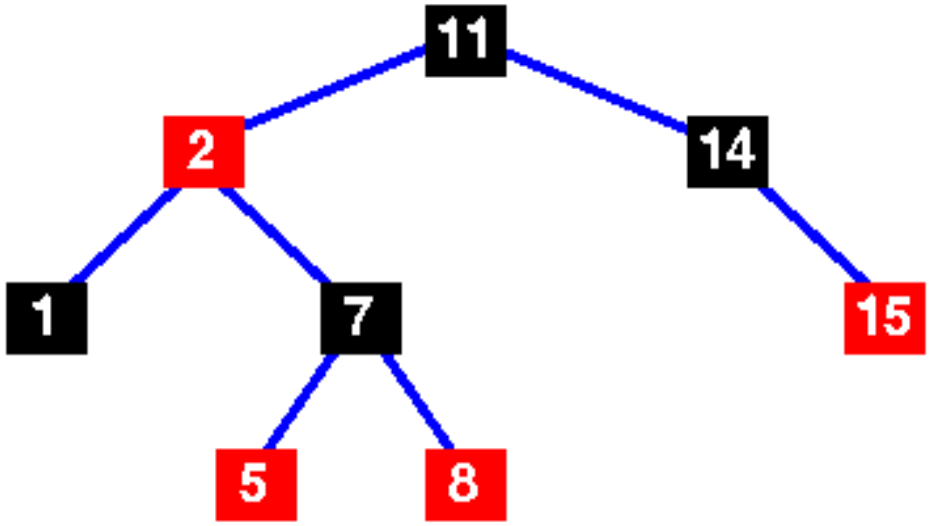


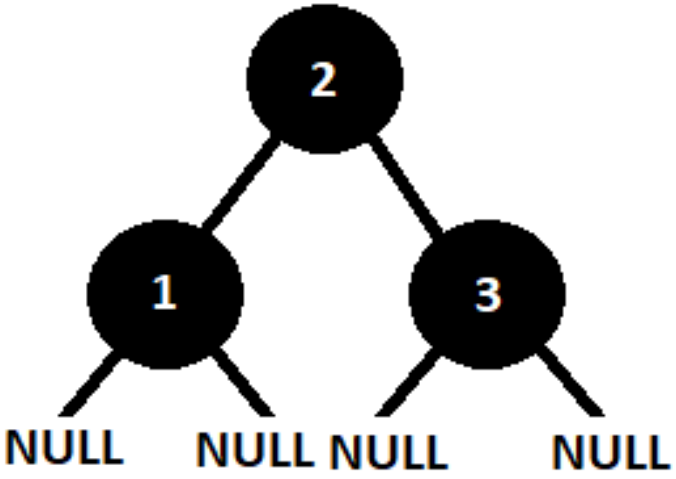
Figure 1. An example of a three-node red-black tree.

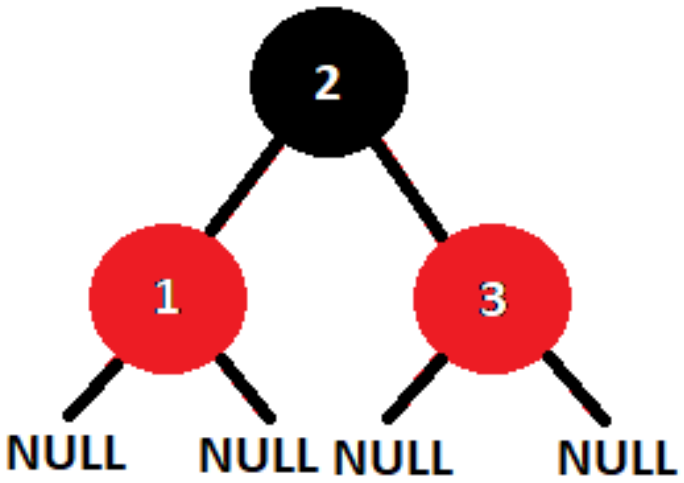
Red-Black Tree Example



Red-Black Tree Example

- Are these valid RB trees?





Red-Black Tree ADT

RedBlackTree:

1. private:
2. root
3. leftRotate(node)
4. rightRotate(node)
5. insertRB(value)
6. rbBalance(node)
7. public:
8. Init()
9. redBlackInsert(value)
10. redBlackDelete(value)
11. search(value)
12. deleteTree()

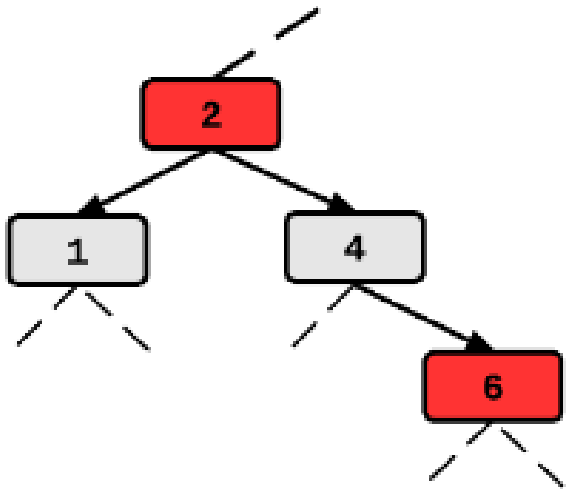
Red-Black Tree Structure

```
struct Node
{
  int data;
  Node *parent;
  Node *left;
  Node *right;
  int color;
};
```

Red-Black Tree Balancing

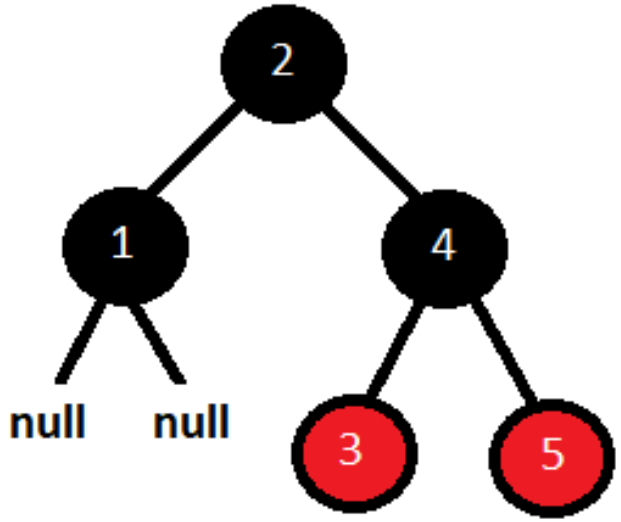
- Use two rules to perform balancing:

1. Recoloring
   - Color red or black
2. Rotation
   - Changes height of the tree
   - Rotate left or right

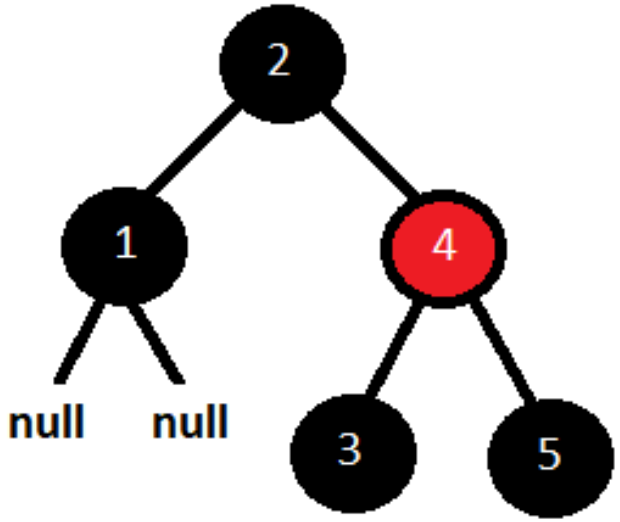


Red-Black Tree Recoloring

- Assume we just inserted node 5

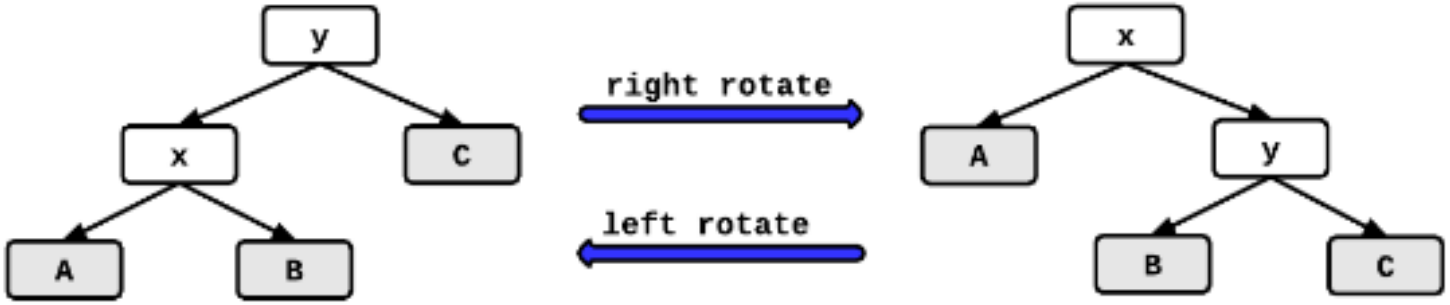


option 1



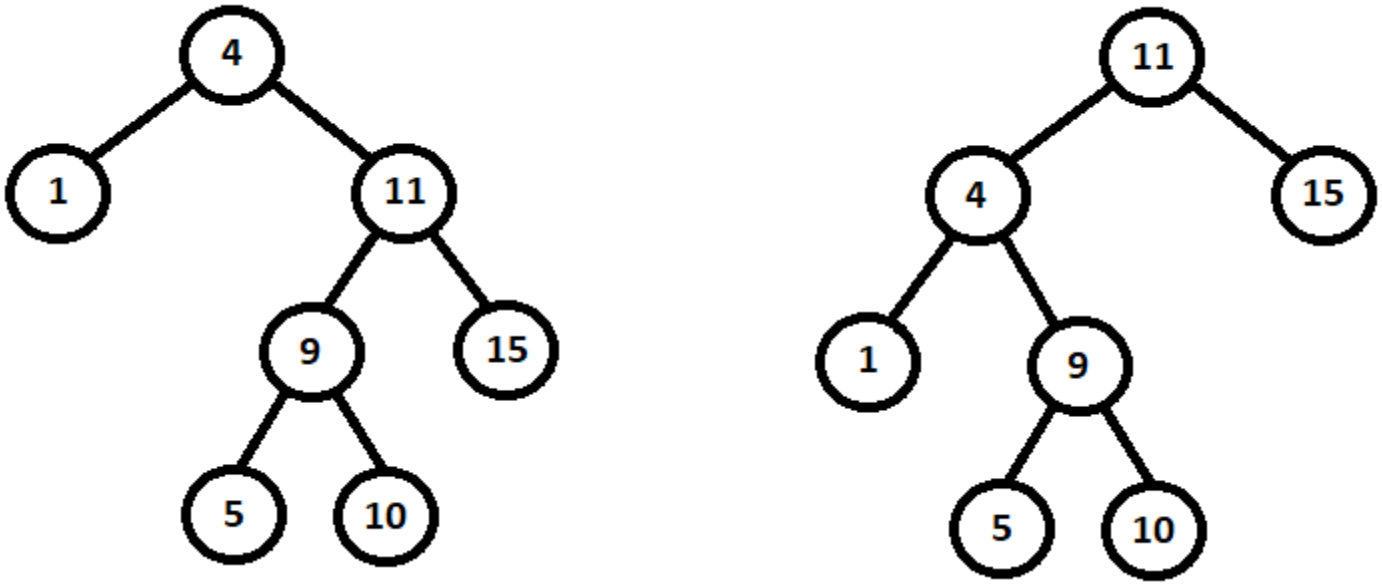
option 2

Red-Black Tree Rotations



Red-Black Tree Rotations

- Left-rotation

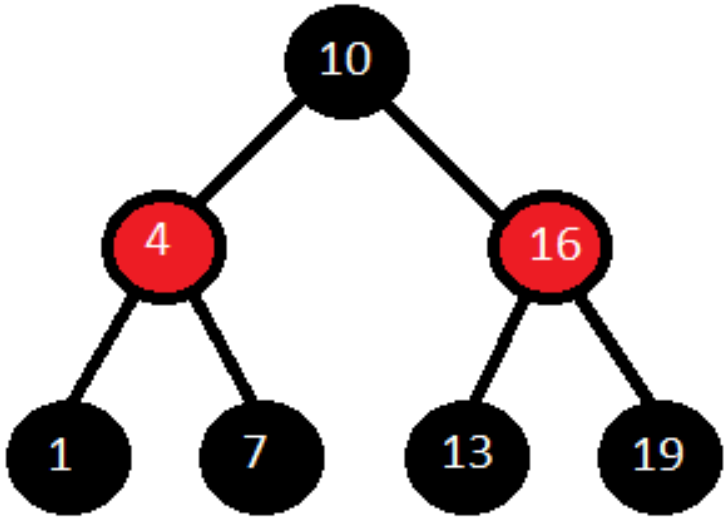


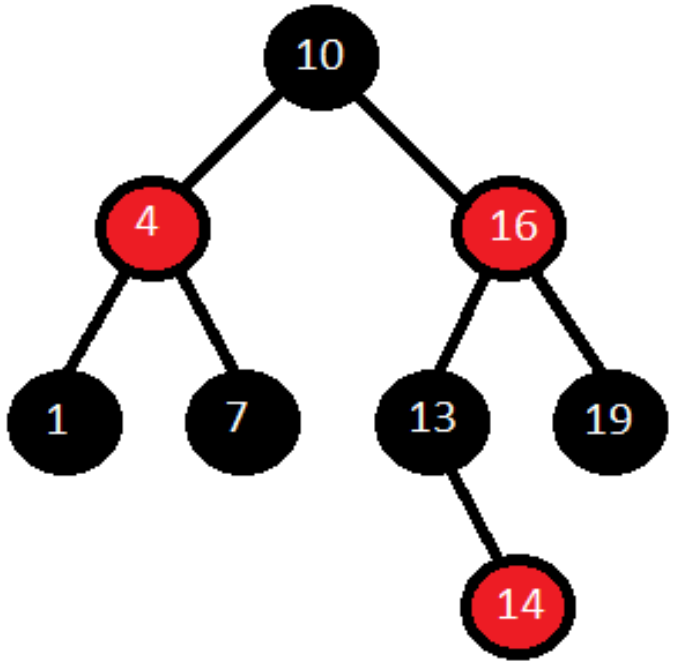
Red-Black Tree Time Compl.

- Rotations are simply O(1) time complexity because we're just changing a couple of pointers
- Coloring an inserted node takes O(1) amount of time since changing a node's color involves a single bit of information.
- Rotating and recoloring nodes in order to fix any rules that may be violated takes O(1) time; but we could have to rotate/recolor our way up the entire tree's height. O(log n) * O(1) = O(log n)

Red-Black Tree Insertion

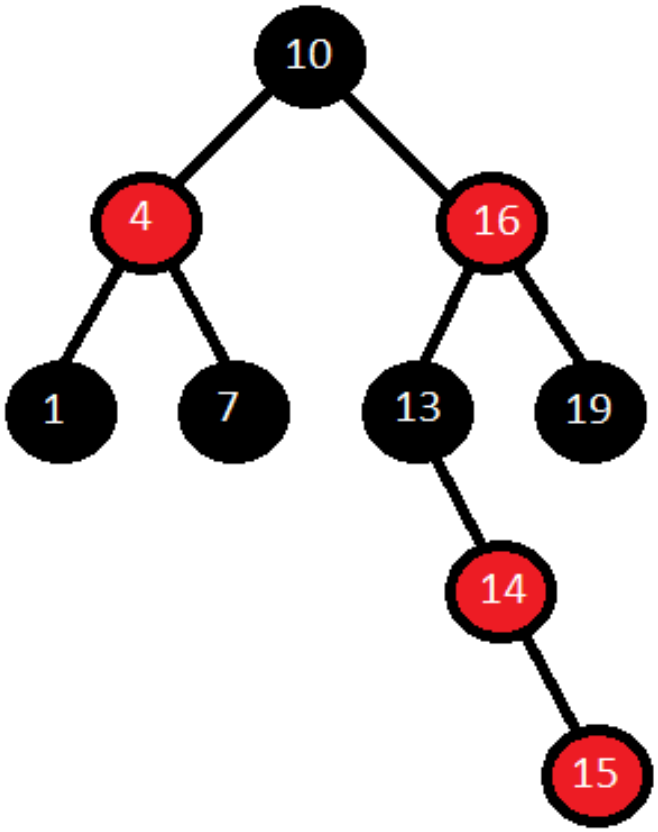
- Assume we want to insert 14

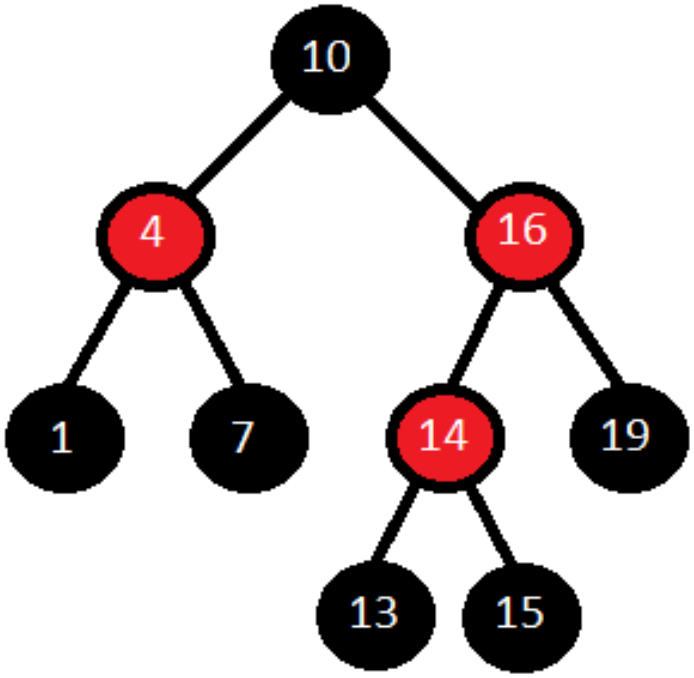




Red-Black Tree Insertion

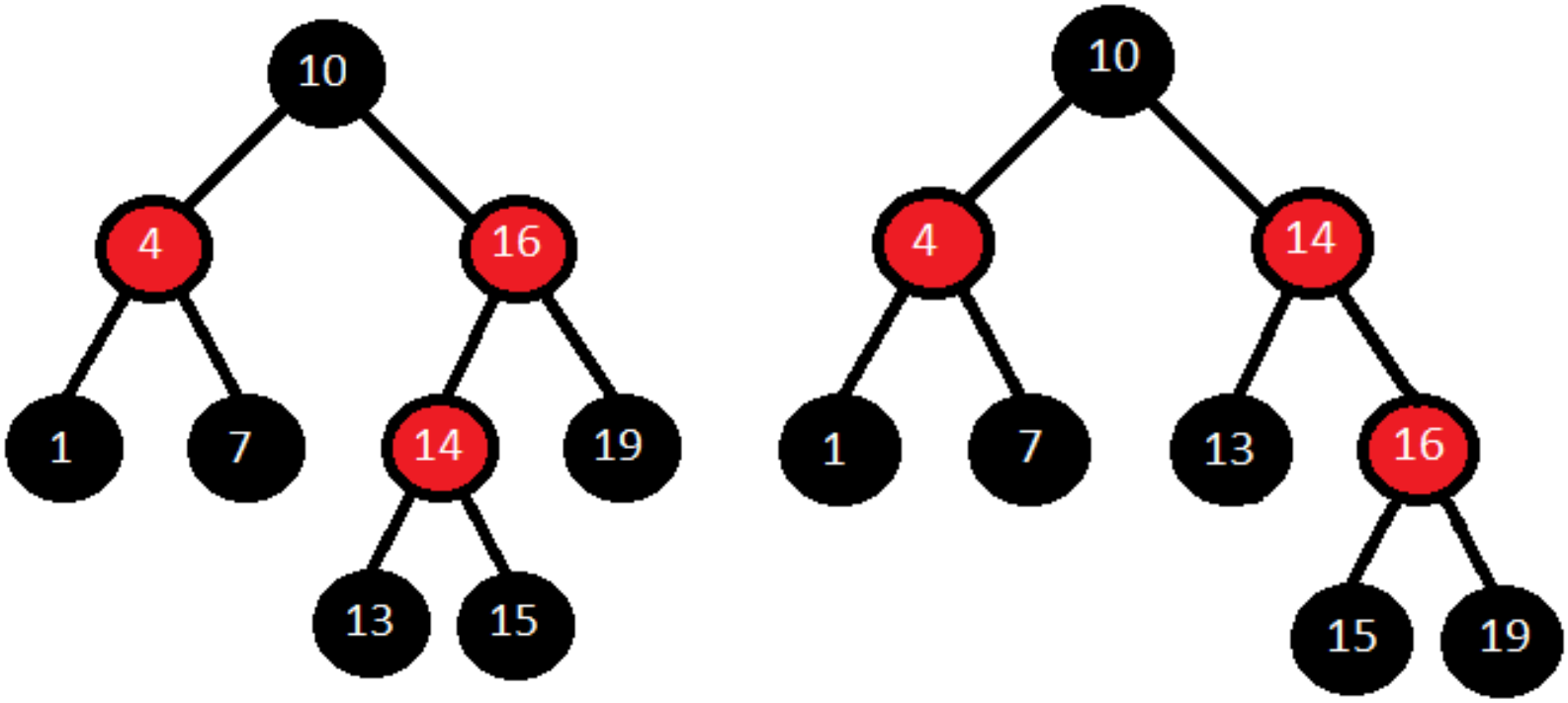
- Now insert 15





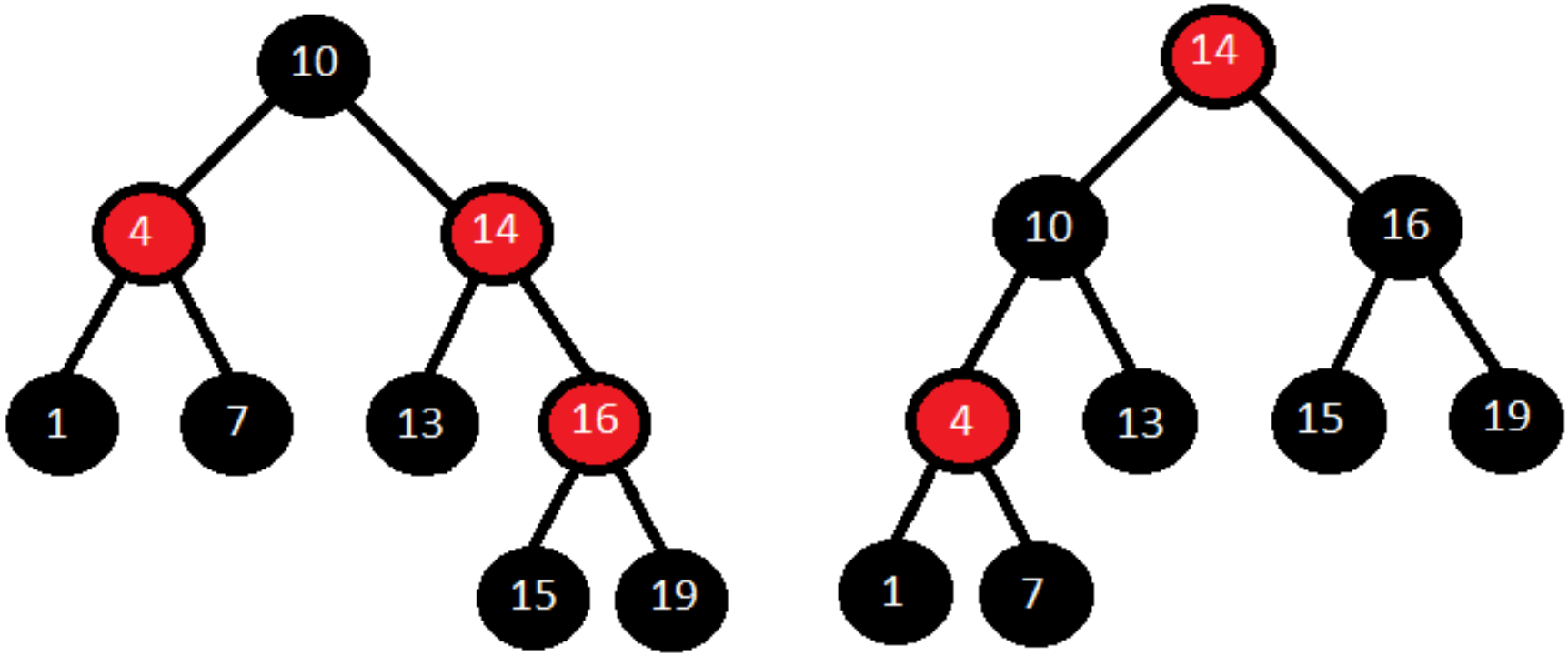
Red-Black Tree Insertion

- Now apply a right rotation



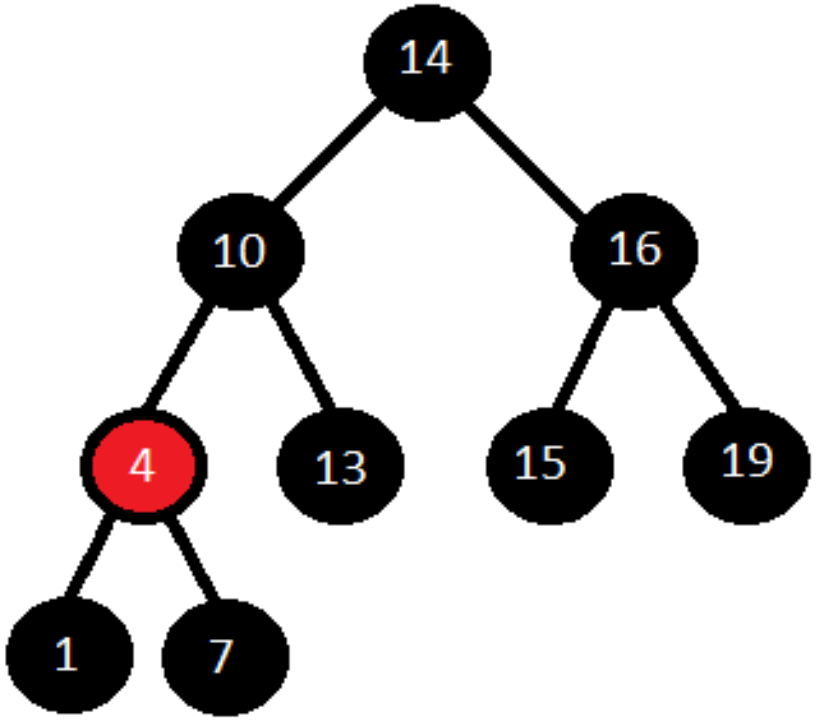
Red-Black Tree Insertion

- …immediately followed by a left rotation



Red-Black Tree Insertion

- One last thing…color the root black



Red-Black Tree Insertion

- There are three changes to the BST insert operation needed to support a red-black tree.

- In a red-black tree:
  1. Replace all instances of NULL in the BST *insert()* algorithm (Algorithm 9.3) with the sentinel node *nullNode*. This change sets the parent of the root to *nullNode* and the left and right children of a new node to *nullNode*.
  2. Set the color of the new node to red.
  3. Resolve any violation of the red-black properties using tree balancing.

Red-Black Tree Insertion

- If *x* is a node added to a red-black tree, then the initial conditions on *x* are:
  - x.color = red
  - x.leftChild =nullNode
  - x.rightChild = nullNode

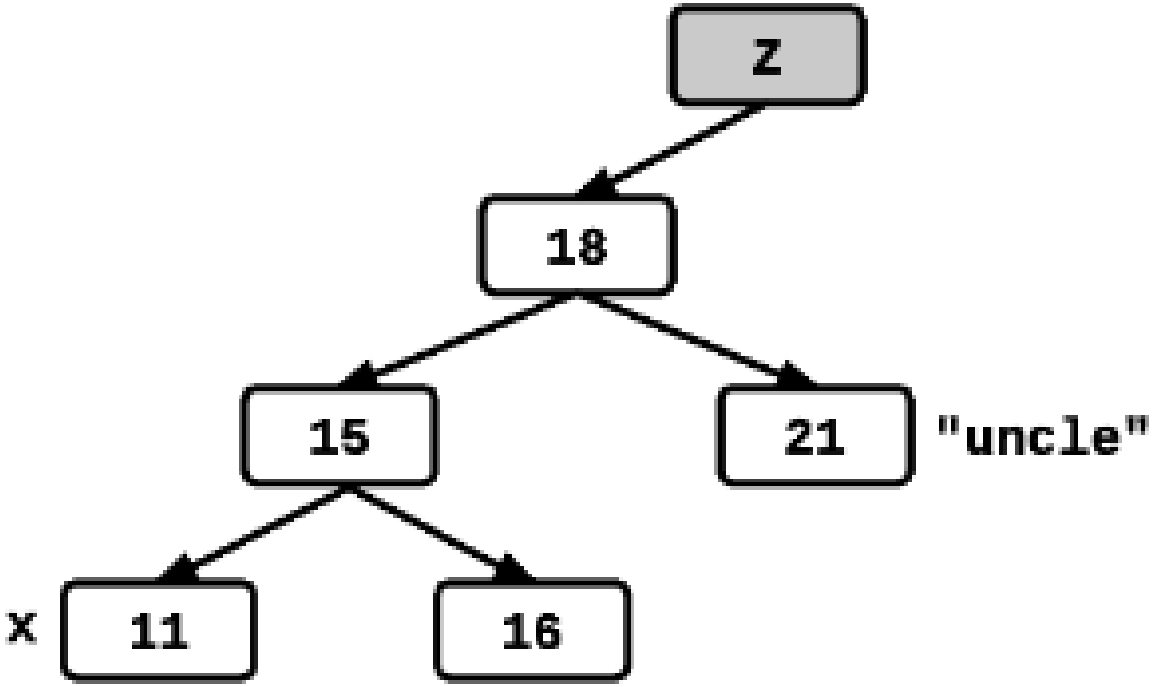
- When a node is added to the tree, the two properties that can be violated are.
  - The root must be black.
  - The children of a red node must be black.

- **Note: Both violations are possible because a new node is initially colored red.**

Red-Black Tree Insertion

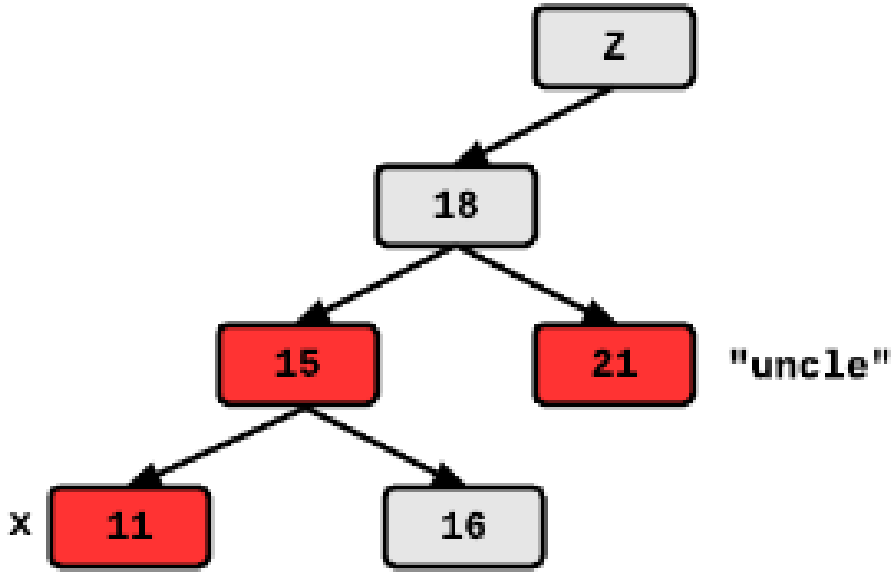
- There are six possible configurations that a red-black tree can take on when a new node is inserted into the tree. Three configurations are symmetric to the other three depending on whether the parent of the new node is the left or right child of its parent.
- The steps needed to rebalance the tree depend on the color of the new node’s "uncle" node.

Red-Black Tree Insertion



Red-Black Tree Insertion

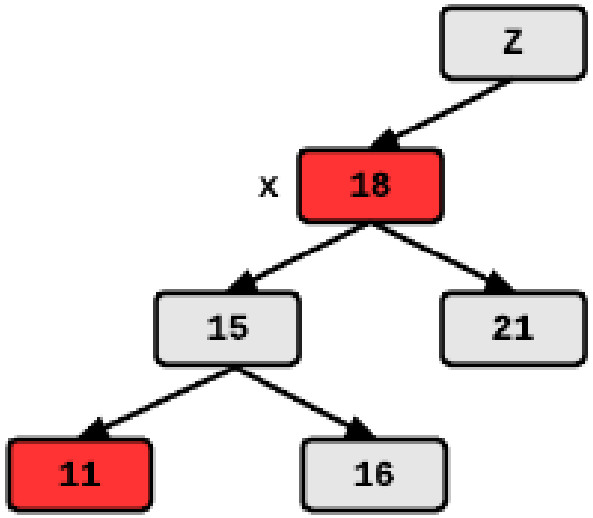
- **Case 1:** The “uncle” node is red.
- If the *parent.parent.rightChild* of the new node is red then *parent* of the new node is also red, and the *parent.parent* of the new node is black.



Red-Black Tree Insertion

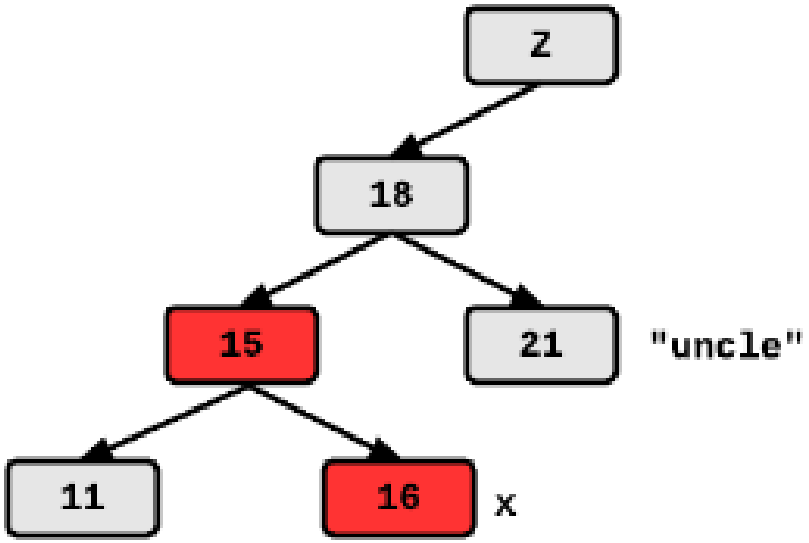
- **Steps to resolve a Case 1 violation in the tree:**

1. Recolor both the *parent* and the "uncle" of the new node to be black, and recolor the *parent.parent* of the new node to be red. This recoloring resolves the violation up to the *parent.parent* level in the tree.
2. Move up two levels in the tree by setting *x* = *x.parent.parent.* The Figure shows the red-black tree after the violations have been resolved. The *x* in the Figure points to the node that would be recolored next, if additional iterations of recoloring were necessary.
3. Repeat Steps 1 and 2 until *x* is the root of the tree or *x's* parent is black.



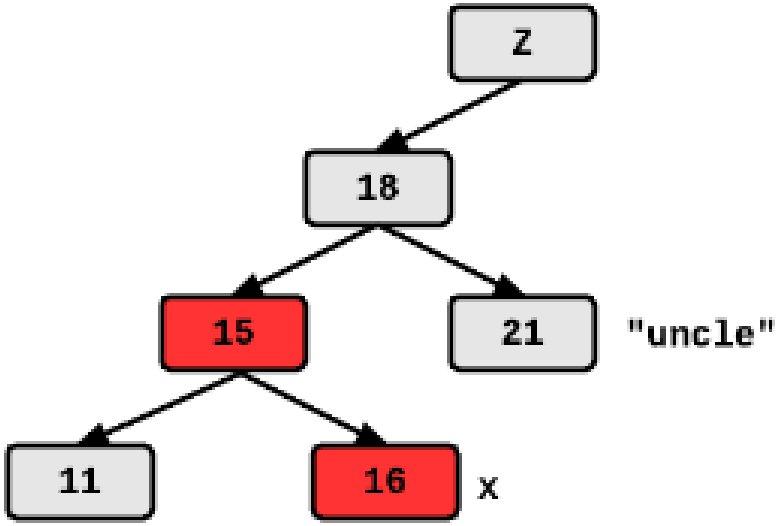
Red-Black Tree Insertion

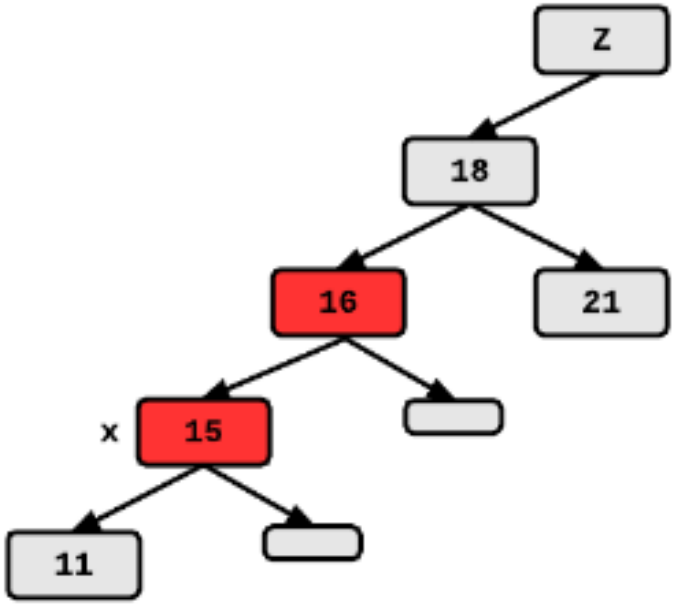
- **Case 2:** The new node is a right child and its uncle is black.
- **Case 3:** The new node is a left child and its uncle is black.



Red-Black Tree Insertion

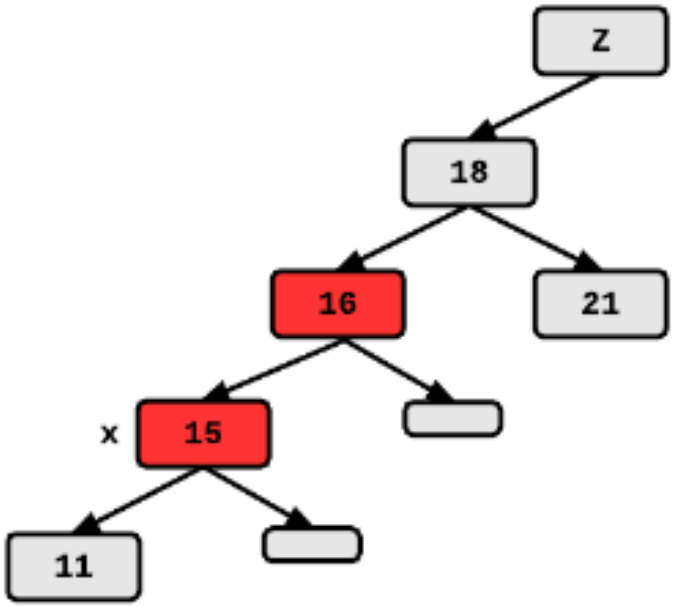
- **Steps to resolve a Case 2 violation in the tree:**
  1. Set *x* = *x.parent.*
  2. Apply the *leftRotate()* algorithm to *x* to convert a Case 2 configuration to a Case 3 configuration. Additional rebalancing can then be applied to resolve the Case 3 violation. The result of the left rotation on the tree in the left Figure is shown in the right Figure. The new node is now a left child of its parent.

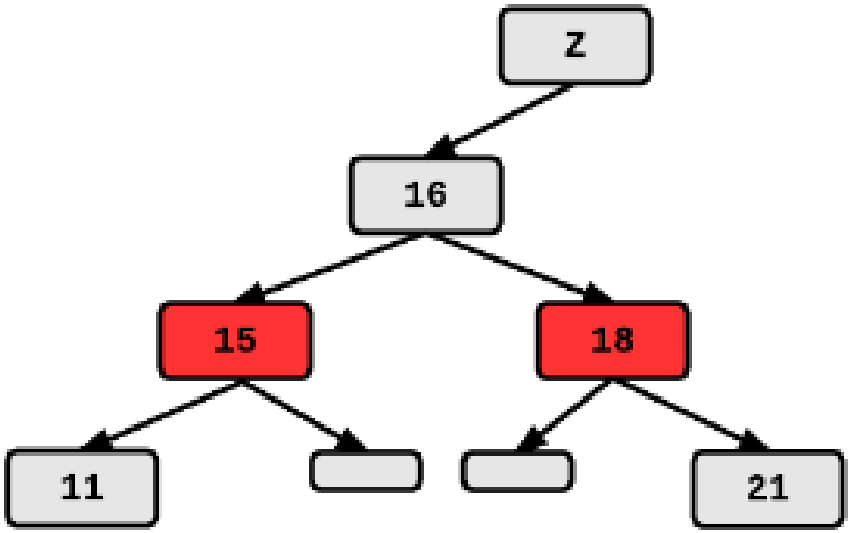




Red-Black Tree Insertion

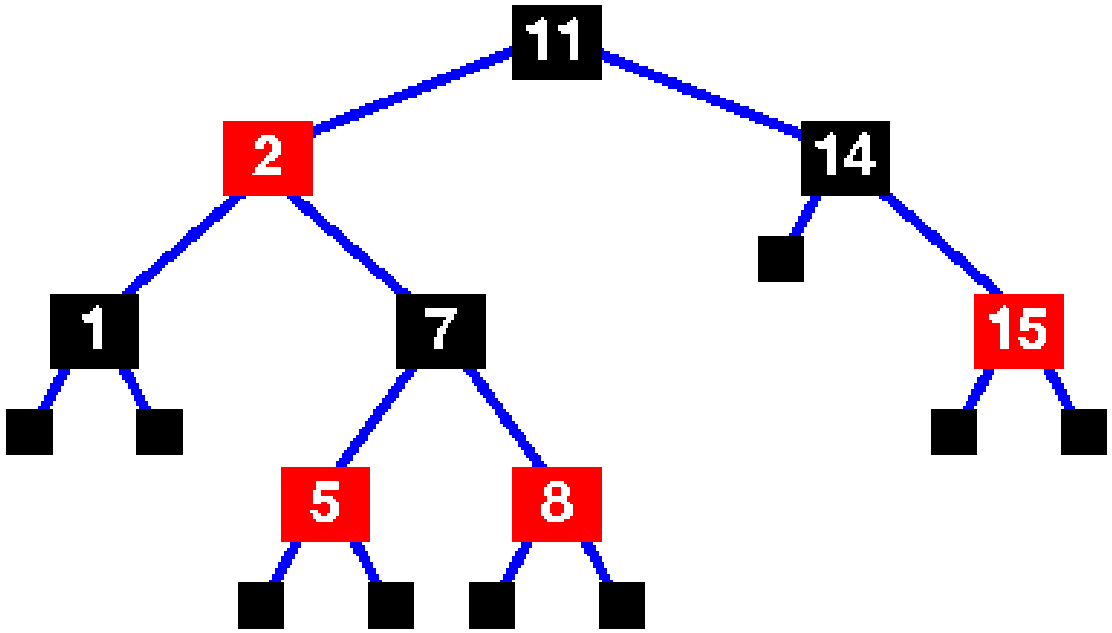
- **Steps to resolve a Case 3 violation in the tree:**
  1. Recolor *x.parent* and *x.parent.parent*.
  2. Apply a right rotation about *x.parent.parent* on the tree in Figure 14 to get the tree in





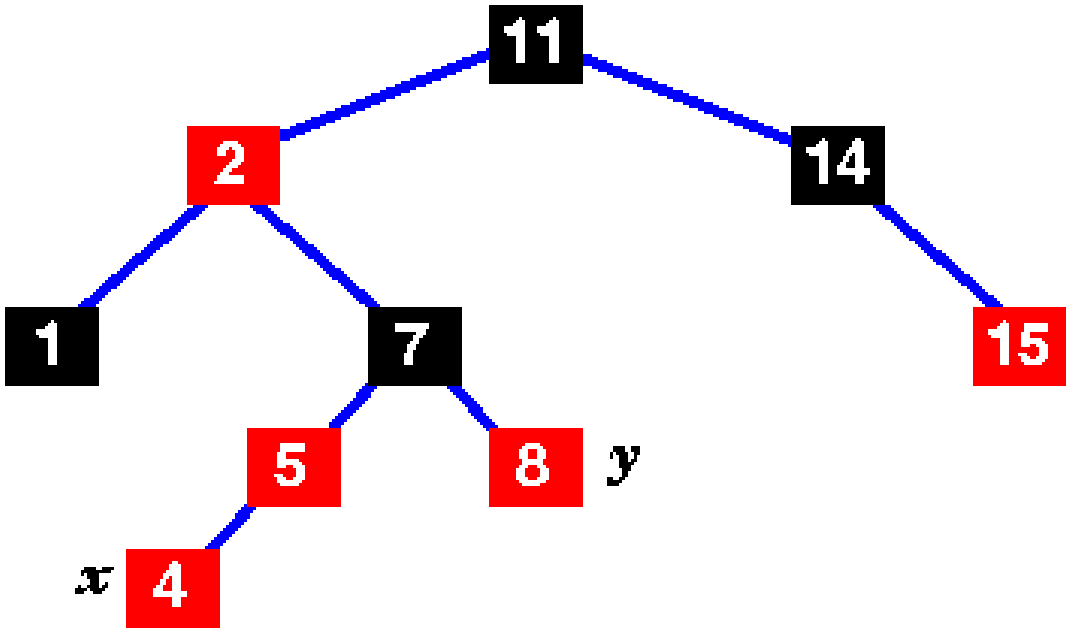
Red-Black Tree Insertion

- Another example

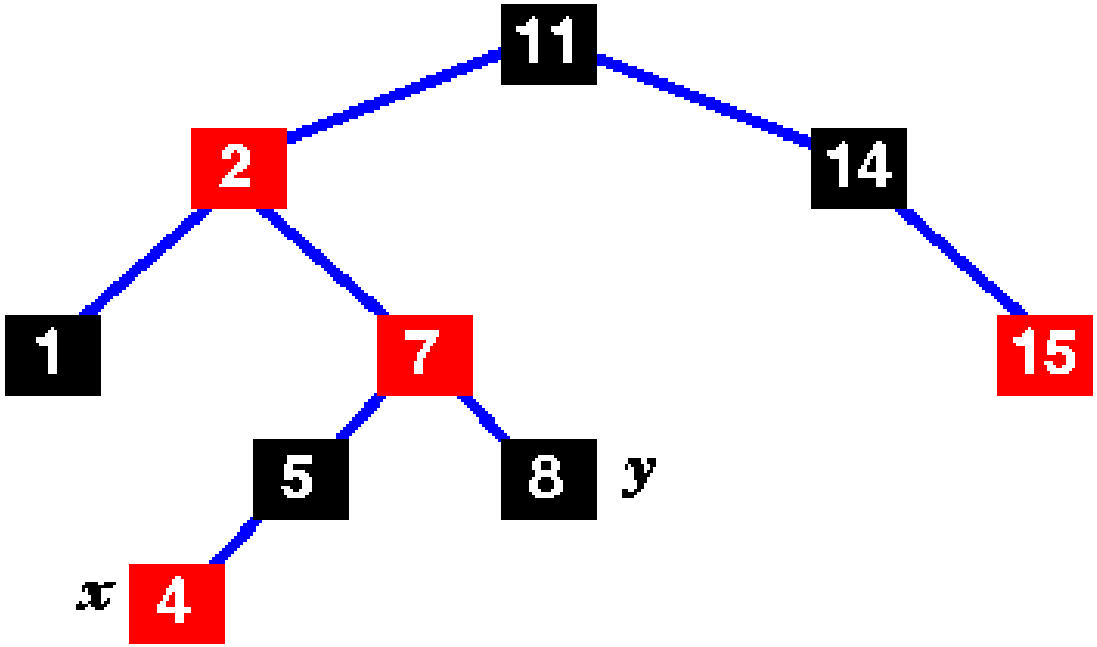


Red-Black Tree Insertion

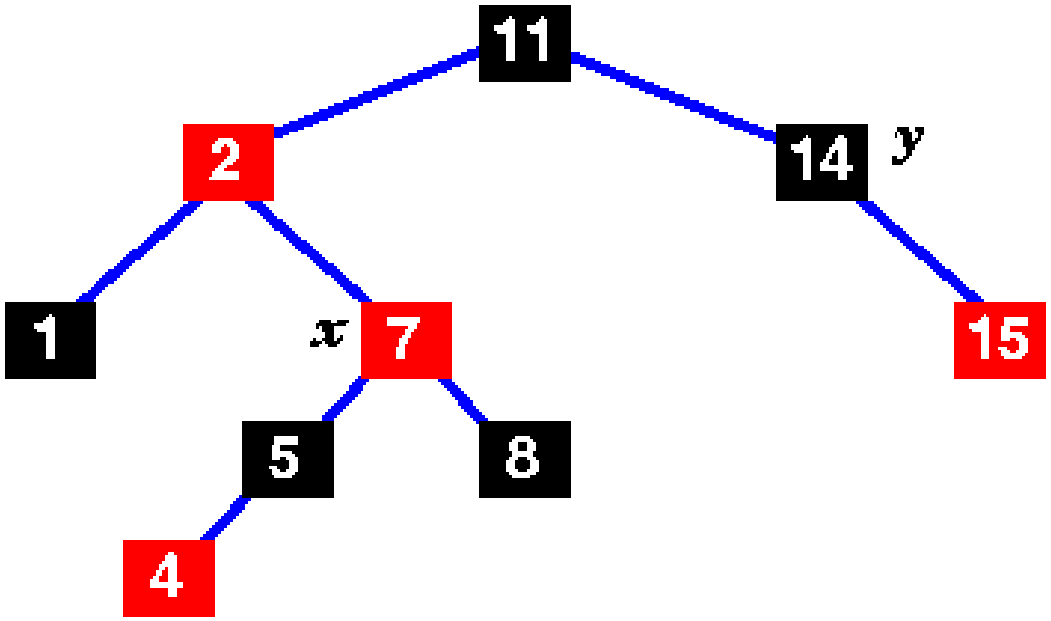
- Which case is violated?



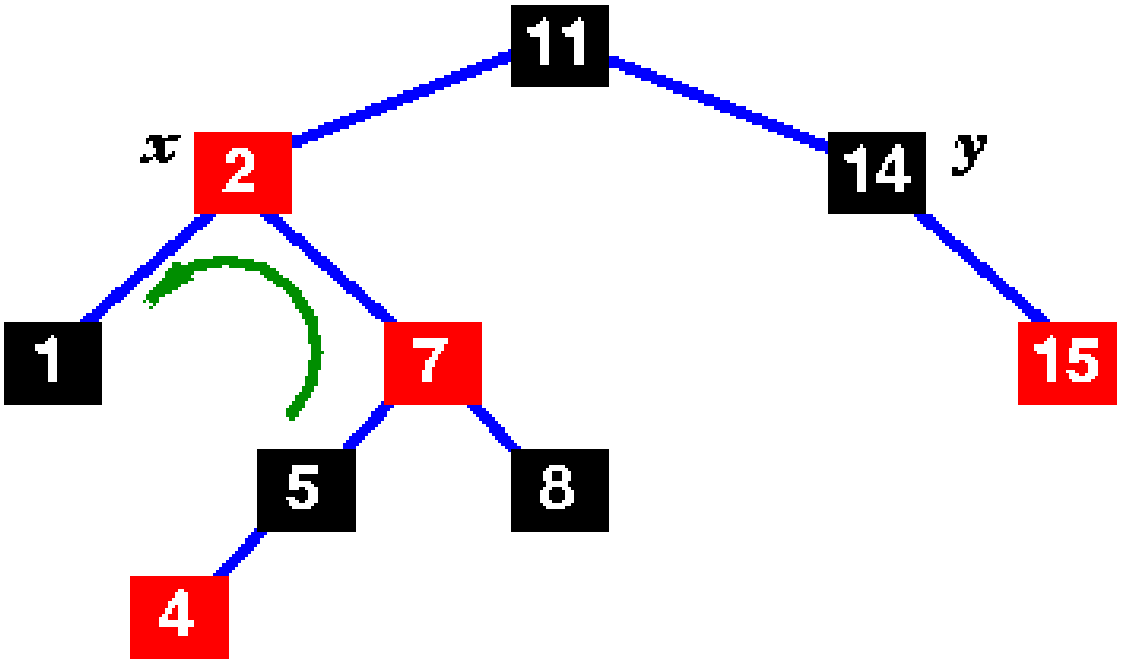
Red-Black Tree Insertion



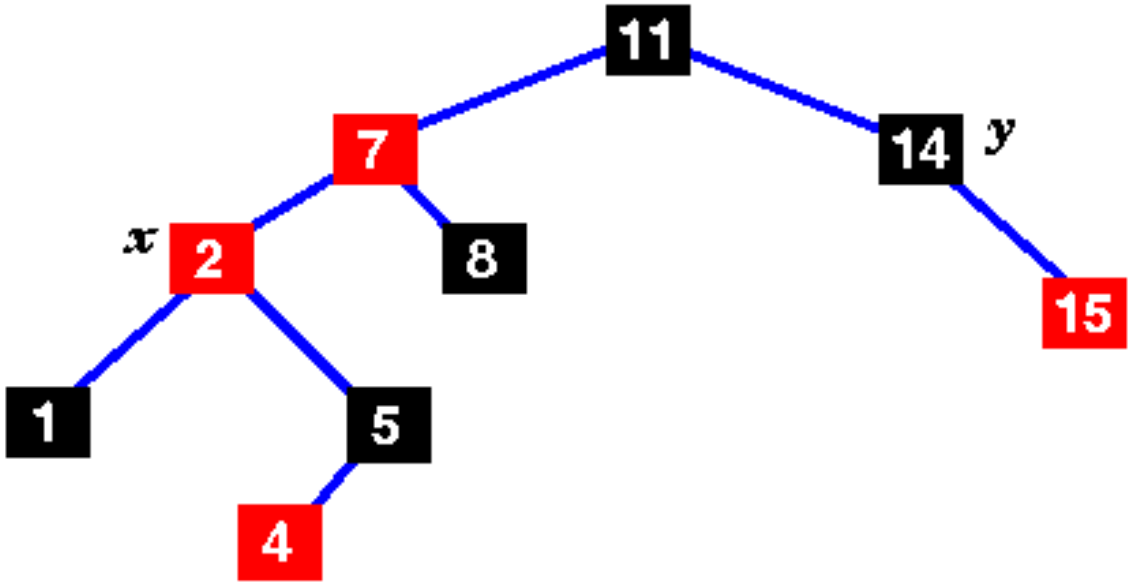
Red-Black Tree Insertion



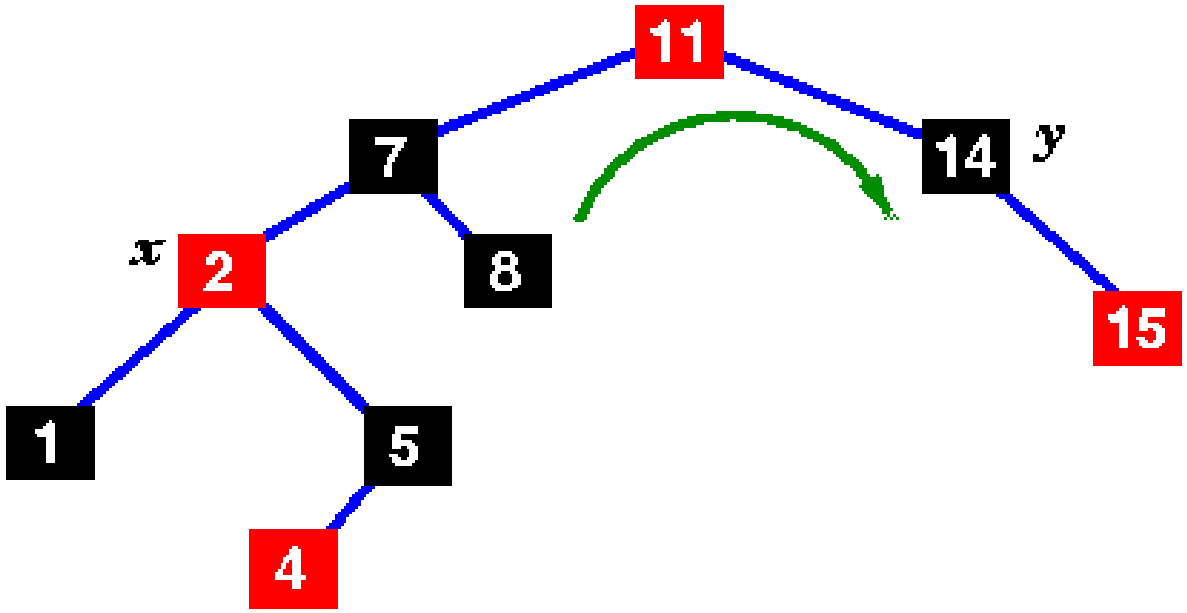
Red-Black Tree Insertion



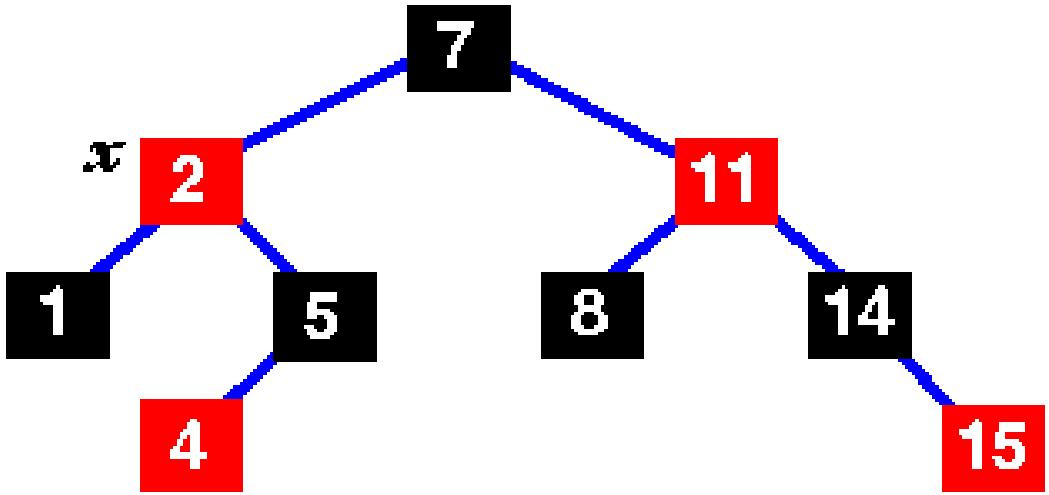
Red-Black Tree Insertion



Red-Black Tree Insertion



Red-Black Tree Insertion

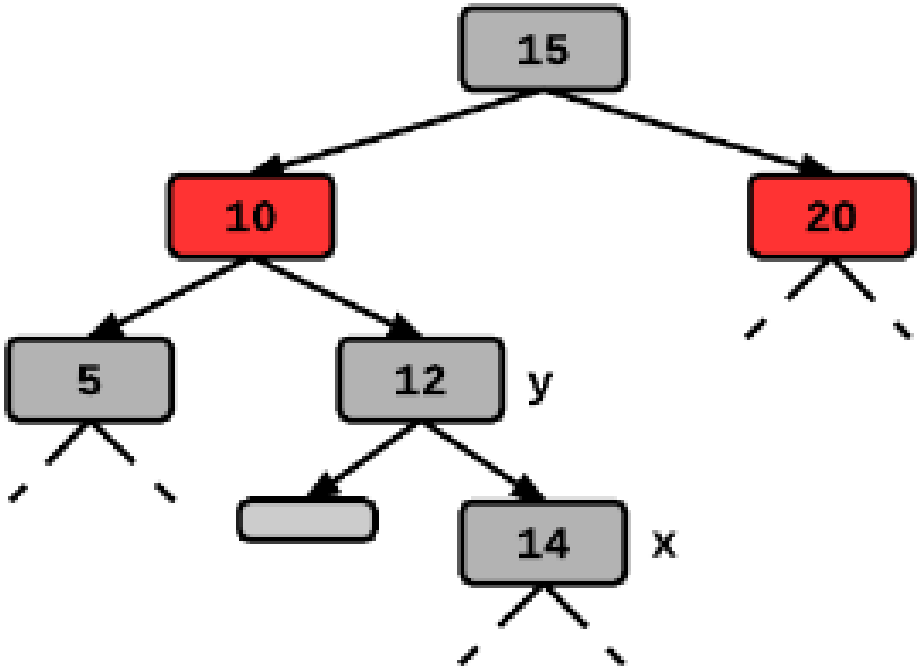


Red-Black Tree Deletion

- The rules for deleting a node from a red-black tree are the same as the rules for deleting a node from a regular BST:
  1. If the node has no children - delete the node.
  2. If the node has one child - replace the node with its remaining child.
  3. If the node has two children - replace the node with the minimum node in its right branch.

Red-Black Tree Deletion

- What happens if we delete node 10?

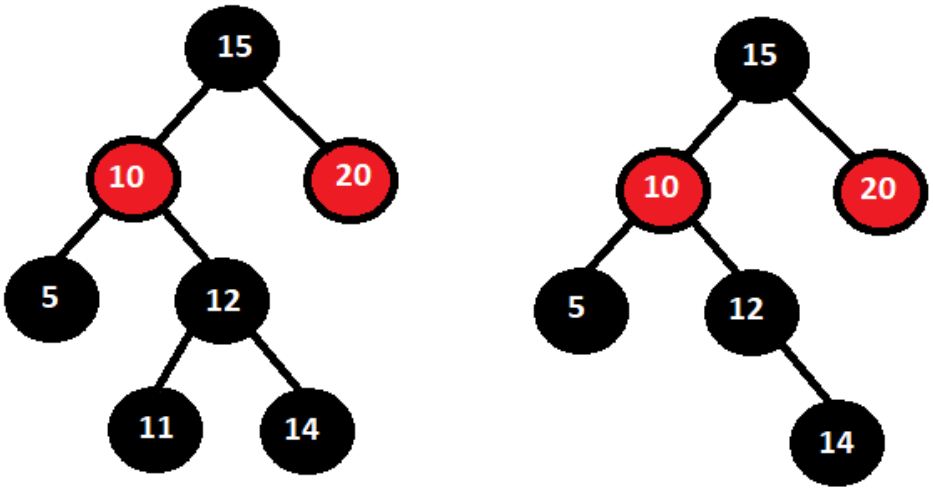


Red-Black Tree Deletion

- We'll focus on a couple of algorithms:
  - redBlackDelete() is broken down into cases for 0, 1, or 2 children and whether the node to delete is the root of the tree.
  - rbBalance() is an additional algorithm to rebalance the tree after the deletion.

Red-Black Tree Deletion

```
redBlackDelete(value)                    // Delete node 11 (value=11)
1.    node = search(value)
2.    nodeColor = node.color
3.    if(node != root)
4.        if(node.leftChild == nullNode and node.rightChild == nullNode) //no children
5.            node.parent.leftChild = nullNode
6.            x = node.parent.leftChild
```



Red-Black Tree Deletion

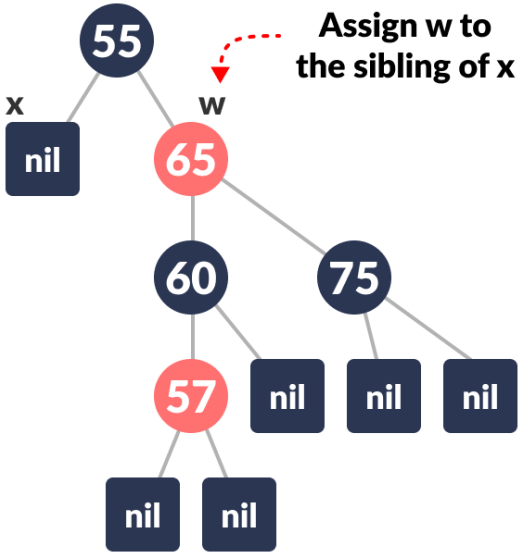
Continue from Friday's lecture

Red-Black Tree Deletion

- Summary of the 3 deletion rules:
  - Case 1: x is a red node
    - *In this case, we simply delete x since deleting a red node doesn't violate any property.*
  - Case 2: x has a red child
    - *We replace x by its red child and change the color of the child to red. This way we retain the property 5.*
  - Case 3: x is a black node **(we are focusing on this case)**

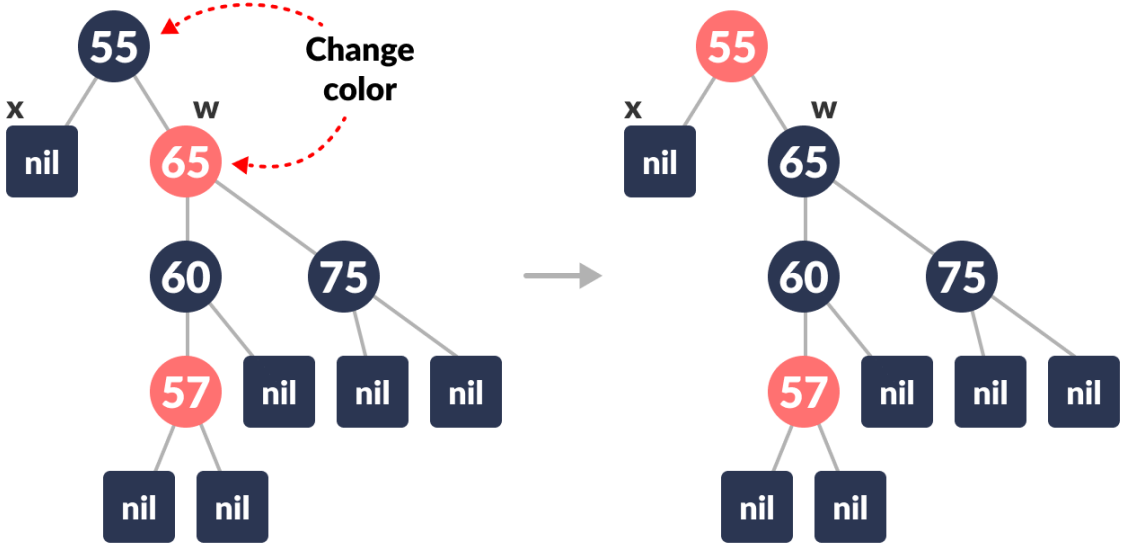
Red-Black Tree Deletion

- rbBalance()
  - Case 3: Deleting a black node violates property 5. In order to maintain property 5, we add an extra black node to the deleted node and call it a ‘double black’ node. Now we need to convert this double black to a single black node.
  - Do the following while x is not the root of the tree and the color of x is BLACK
    - while x != root and x.color == BLACK
  - If x is the left child of its parent then,
    - Assign w to the sibling of x



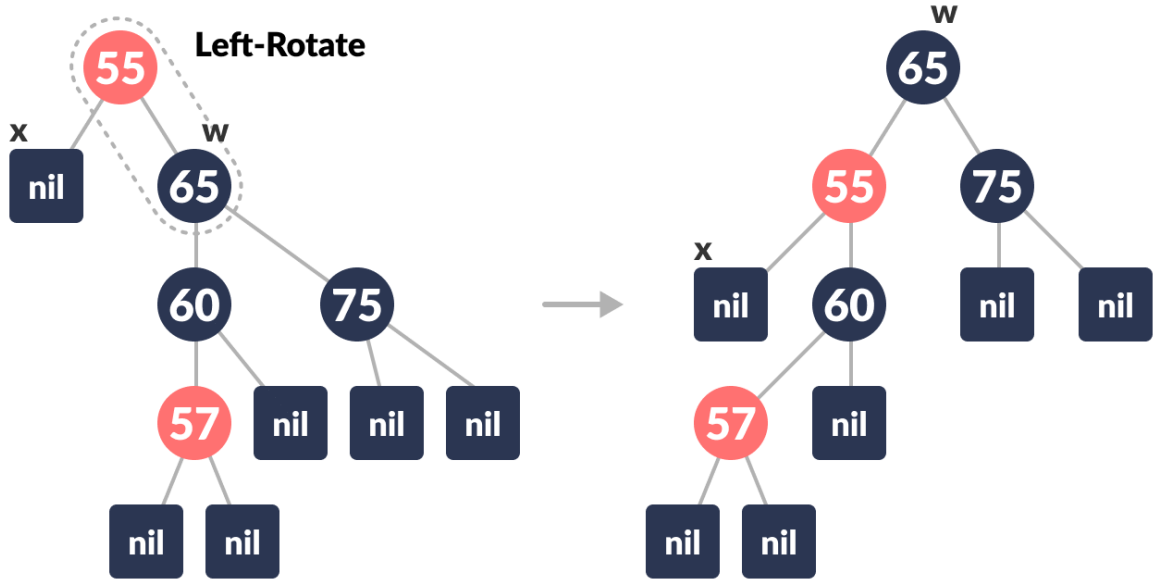
Red-Black Tree Deletion

- rbBalance()
  - Case 3.1: x's sibling w is red
    - **Set the color of the right child of the parent of x as BLACK.**
    - **Set the color of the parent of x as RED.**
    - Left-Rotate the parent of x.
    - Assign the rightChild of the parent of x to w.



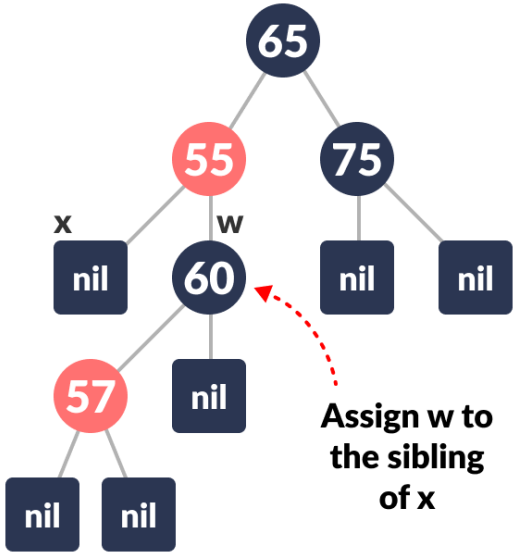
Red-Black Tree Deletion

- rbBalance()
  - Case 3.1: x's sibling w is red
    - Set the color of the right child of the parent of x as BLACK.
    - Set the color of the parent of x as RED.
    - **Left-Rotate the parent of x.**
    - Assign the rightChild of the parent of x to w.



Red-Black Tree Deletion

- rbBalance()
  - Case 3.1: x's sibling w is red
    - Set the color of the right child of the parent of x as BLACK.
    - Set the color of the parent of x as RED.
    - Left-Rotate the parent of x.
    - **Assign the rightChild of the parent of x to w.**

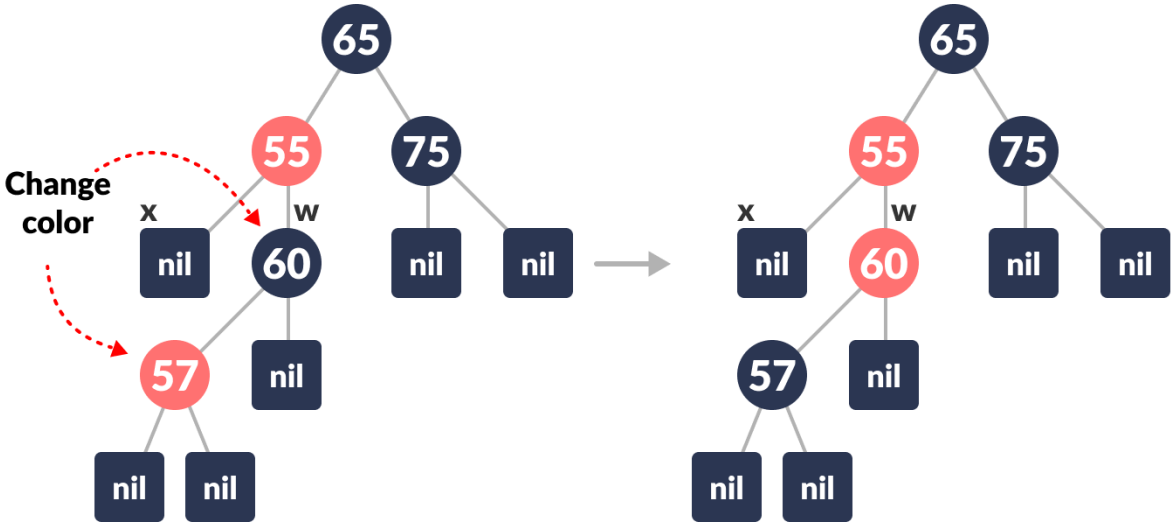


Red-Black Tree Deletion

- rbBalance()
  - Case 3.2: x's sibling w is black; both children of w are black.
    - Set the color of w as RED.
    - Assign the parent of x to x.
  - **Case 3.2 does not apply to our example so instead, I am showing a separate, generic example below of what this case would look like.**
    - ***If new x is red, change it to black; else, repeat.***
      - **Note: the color of B could be red or black**

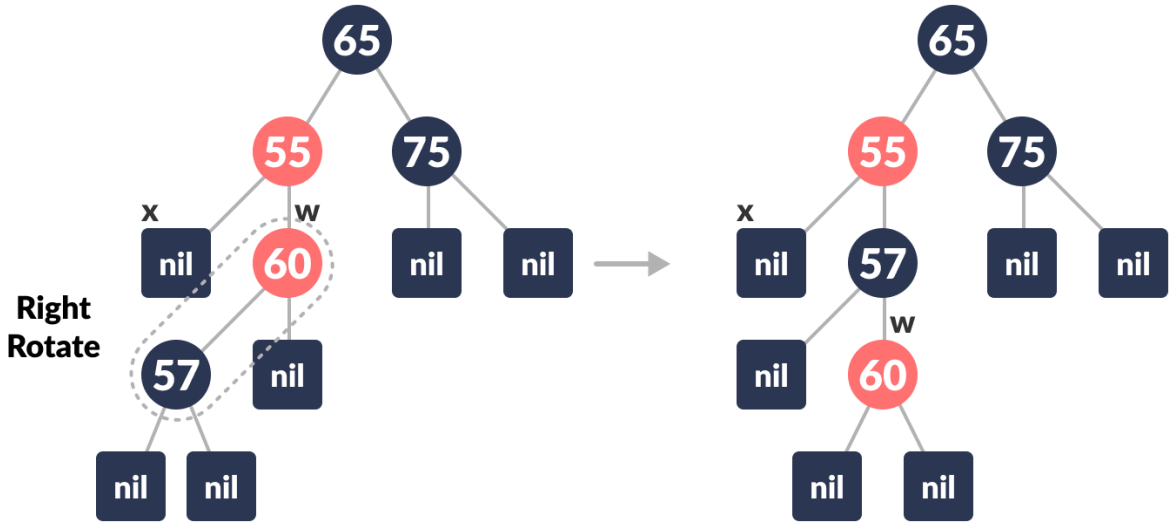
Red-Black Tree Deletion

- rbBalance()
  - Else if the color of the rightChild of w is BLACK, we have Case 3.3:
  - Case 3.3: x's sibling w is black; w's left child is red, w's right child is black
    - **Set the color of the leftChild of w as BLACK.**
    - **Set the color of w as RED.**
    - Right-Rotate w.
    - Assign the rightChild of the parent of x to w.



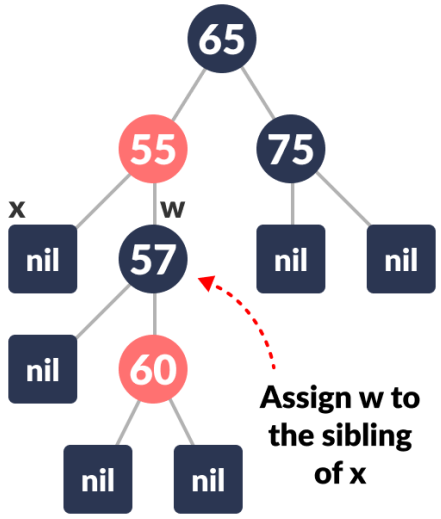
Red-Black Tree Deletion

- rbBalance()
  - Else if the color of the rightChild of w is BLACK, we have Case 3.3:
  - Case 3.3: x's sibling w is black; w's left child is red, w's right child is black
    - Set the color of the leftChild of w as BLACK.
    - Set the color of w as RED.
    - **Right-Rotate w.**
    - Assign the rightChild of the parent of x to w.



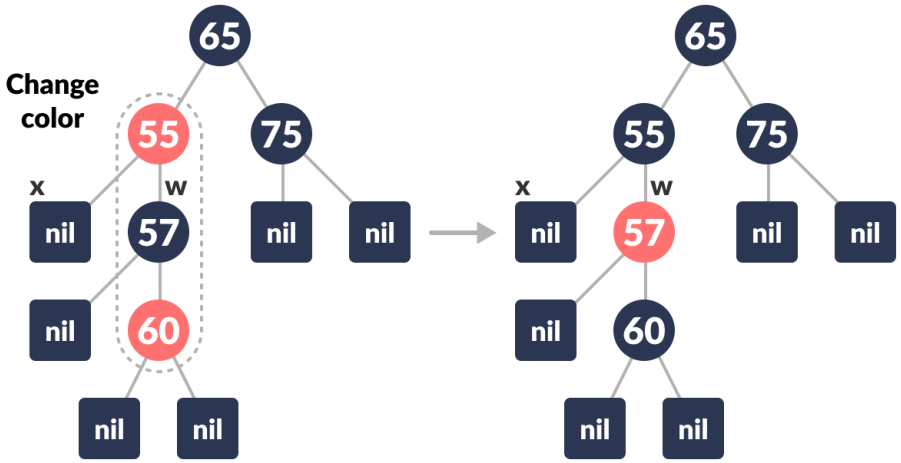
Red-Black Tree Deletion

- rbBalance()
  - Else if the color of the rightChild of w is BLACK, we have Case 3.3:
  - Case 3.3: x's sibling w is black; w's left child is red, w's right child is black
    - Set the color of the leftChild of w as BLACK.
    - Set the color of w as RED.
    - Right-Rotate w.
    - **Assign the rightChild of the parent of x to w.**



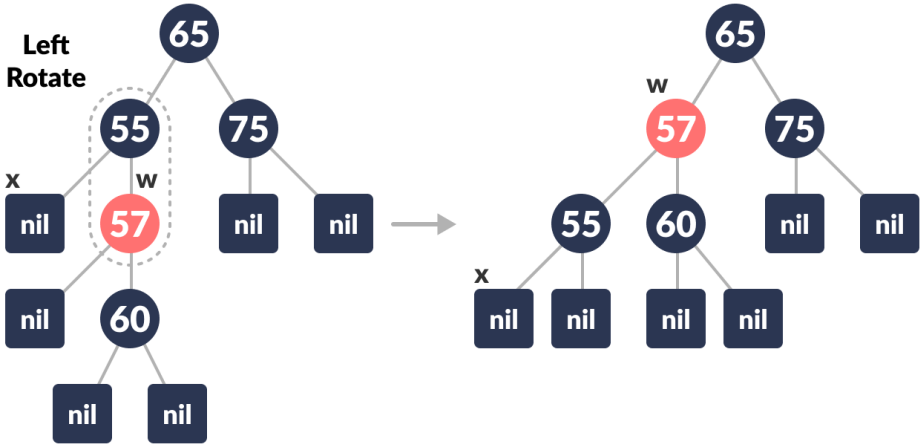
Red-Black Tree Deletion

- rbBalance()
  - If any of the above cases do not occur, then do the following.
  - Case 3.4: x's sibling w is black; w's right child is red
    - **Set the color of w as the color of the parent of x.**
    - **Set the color of the parent of parent of x as BLACK.**
    - **Set the color of the right child of w as BLACK.**
    - Left-Rotate the parent of x.
    - Set x as the root of the tree.



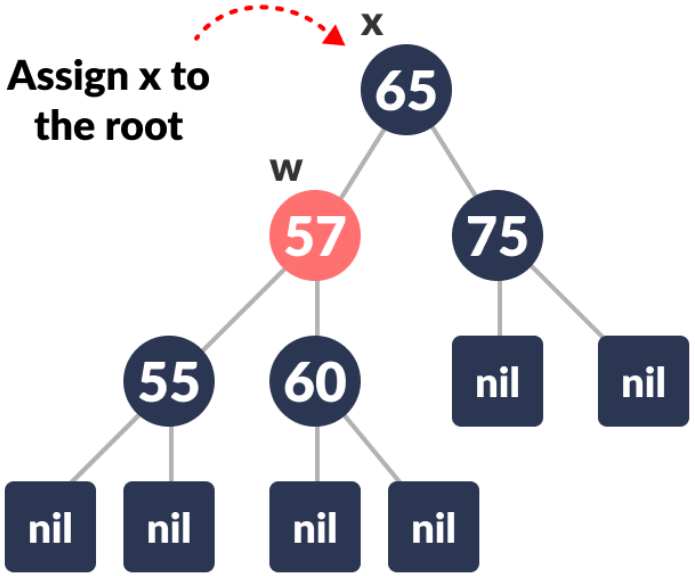
Red-Black Tree Deletion

- rbBalance()
  - If any of the above cases do not occur, then do the following.
  - Case 3.4: x's sibling w is black; w's right child is red
    - Set the color of w as the color of the parent of x.
    - Set the color of the parent of parent of x as BLACK.
    - Set the color of the right child of w as BLACK.
    - **Left-Rotate the parent of x.**
    - Set x as the root of the tree.



Red-Black Tree Deletion

- rbBalance()
  - If any of the above cases do not occur, then do the following.
  - Case 3.4: x's sibling w is black; w's right child is red
    - Set the color of w as the color of the parent of x.
    - Set the color of the parent of parent of x as BLACK.
    - Set the color of the right child of w as BLACK.
    - Left-Rotate the parent of x.
    - **Set x as the root of the tree.**



Red-Black Tree Deletion

- rbBalance()
  - When we started this example, we said:
    if x is the left child of tis parent then,
    assign w to the sibling of x

  - The else condition for the above pseudocode is
    Else same as above with right changed to left and vice versa

Red-Black Tree Deletion

- rbBalance()
  - The final step of rbBalance() is:
    - **Set the color of x as BLACK.**

Red-Black Tree Deletion

```
rbBalance(T, x)
    rbBalance(T, x)
    while x ≠ T.root and x.color == BLACK
       if x == x.parent.left
          s = x.parent.right
          if s.color == RED
             s.color = BLACK // case 3.1
             x.parent.color = RED // case 3.1
             LEFT-ROTATE(T, x.parent) // case 3.1
             s = x.parent.right // case 3.1
          if s.left.color == BLACK and s.right.color == BLACK
             s.color = RED // case 3.2
             x = x.parent //case 3.2
          else if s.right.color == BLACK
                s.left.color = BLACK // case 3.3
                s.color = RED //case 3.3
                RIGHT-ROTATE(T, s) // case 3.3
                s = x.parent.right // case 3.3
             s.color = x.parent.right // case 3.4
             x.parent.color = BLACK // case 3.4
             s.right.color = BLACK // case 3.4
             LEFT-ROTATE(T, x.parent) // case 3.4
             x = T.root
       else (same as then close with “right” and “left” exchanged)
    x.color = BLACK
Set the color of x as BLACK.
```

Questions?