Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Trees III: Balanced Binary Search Trees & Red-Black Trees

""

CS2013: Programming with Data Structures

Balanced Binary Search Trees

The Need for Balancing

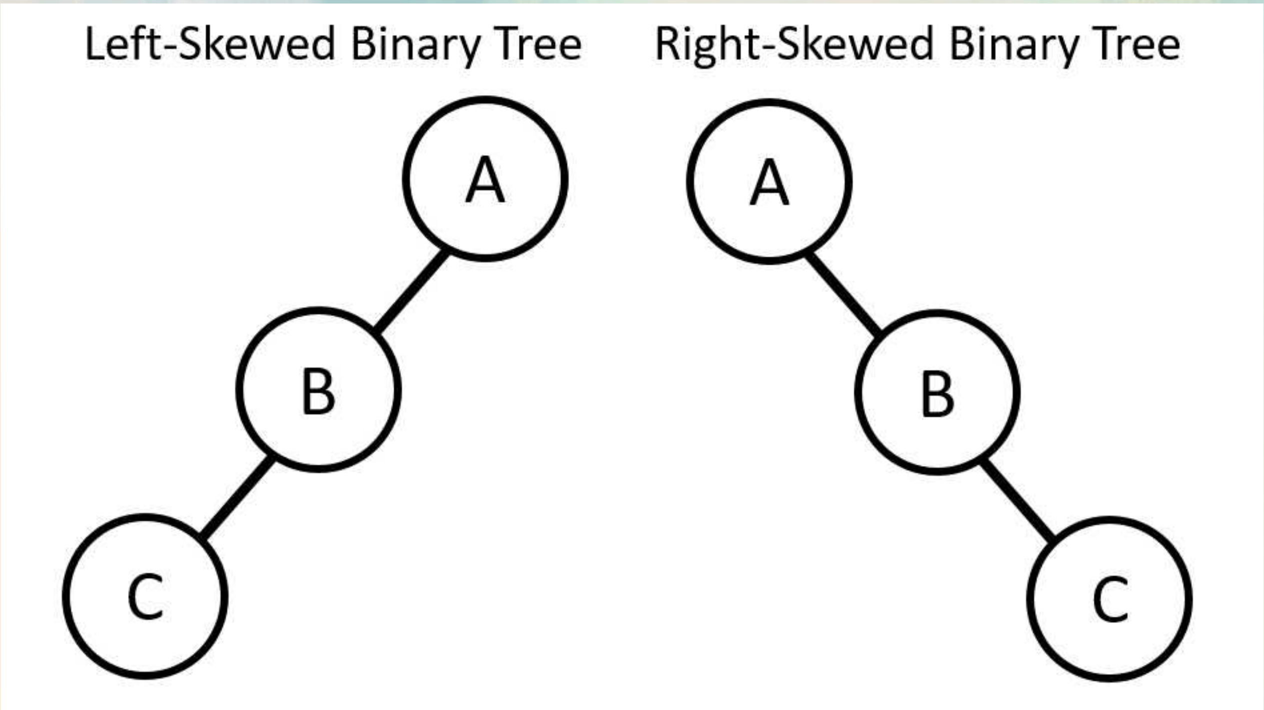
- ***balanced binary tree***: a binary tree which tries to minimize its height regardless of the number of insertion and deletion operations.

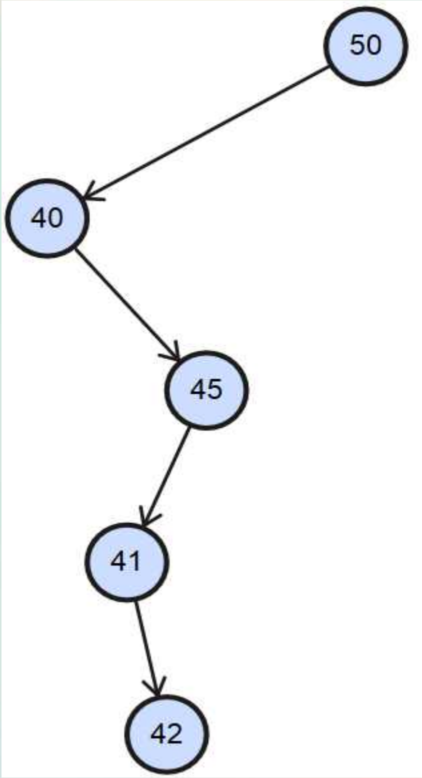
- Why balance?
  - the benefit of BSTs is that you can have a O(logn) performance for many of the algorithms (find(), delete(), insert())
  - in some instances these operations will be O(n)
    - When does this happen?

Skewed Trees

- ***skewed tree:*** a tree in which all the nodes except one have one and only one child.
  - looks more like a linked-list
  - operations become O(n) instead of O(logn)

- Trees can be left-skewed or right-skewed or in a zigzag pattern (skewed).





The Need for Balancing

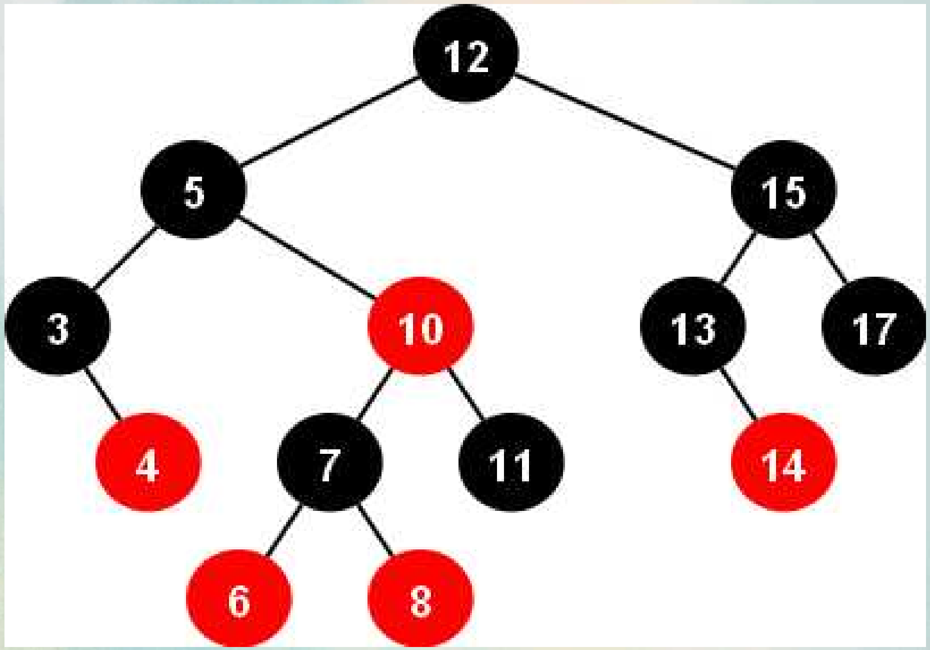
- By implementing a tree that is ***self-balancing***, the tree can correct its height through a series of operations to prevent skewing and keep the height as minimal as possible.

- Examples of Self-Balancing Binary Search Trees:
  - Red-Black Trees
  - AVL (Adelson-Velskii and Landis) Trees
  - Splay Tree
  - 2-3 Tree
  - and others...

Red-Black Trees

Red-Black Trees

- have all the properties of a binary search tree
  - same algorithms for insert(), delete(), and find()



- have these additional properties:
  - 1. Every node is colored red or black.
  - 2. The root is black.
  - 3. If a node is red, then both of its children must be black.
  - 4. Every NIL leaf is black.
    - If your implementation does not have the special NIL leaf, then a child that is NULL is considered to be black.
  - 5. For each node *n,* every path from *n* to a leaf must contain the same number of black nodes.

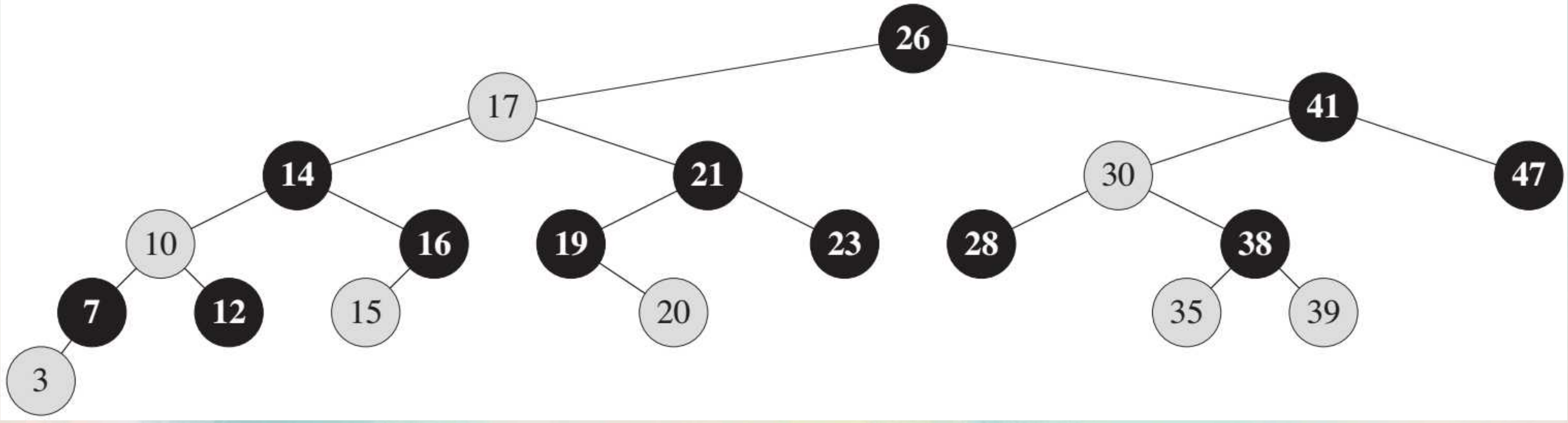
Optional Property and NIL (NULL) Leaves

- Red-Black trees can be implemented with two main designs:
  - With special leaves called the NIL or NULL leaves.
  - Without the special leaves.

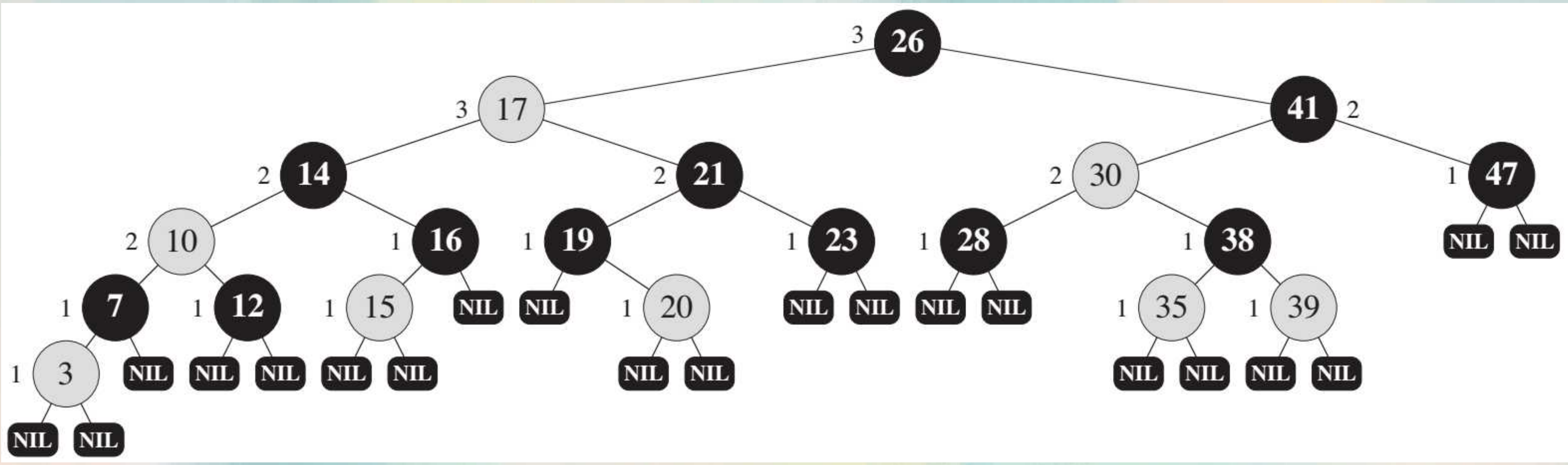
- If the NIL leaves exist the they are special instances of the RBNode class which have a value of **null** and color of black.

- In an implementation without the NIL leaves, a left or right child which has a NULL value is still considered to have a color of black.
  - This is used when you want to find a sibling or uncle and the sibling or uncle is null.
  - We will say the sibling or uncle has a color of black.

Red-Black Tree without NIL Leaves



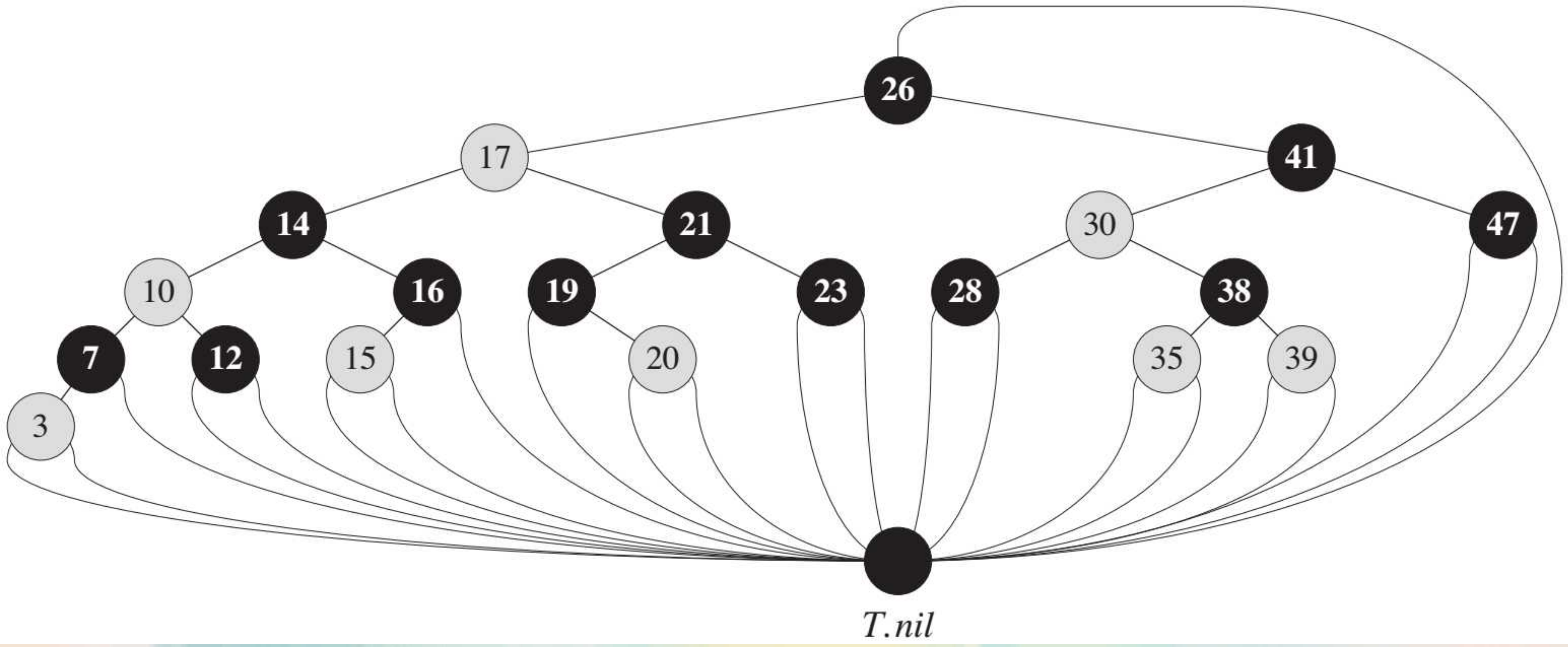
A Red-Black Tree with NIL Leaves



Red-Black Tree with One Instance of NIL

- It would be impractical to create a new instance of a NIL leaf every time a node did not have a child.
  - This would waste memory.

- Instead, you can create a static data field representing a special instance of your Node class that can be referred to by any nodes that don't have a left / right child.



Result of Properties

- These properties enforce that: ***the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf.***

- The tree will be height-balanced.

- The tree will have O(logn) performance even in the worst case.

- In other words, it helps to prevent skew trees whose performance is O(n).

Red-Black Tree Nodes

- A possible Node class for a Red-Black tree might look something like the following:

```
public class RBNode<E extends Comparable<E>>
    protected E data;
    protected char color;
    protected RBNode<E> left = null;
    protected RBNode<E> right = null;
    protected RBNode<E> parent = null;

    public RBNode(E data, char color) {
        this.data = data;
        this.color = color;
    }
```

- Note:
  - we keep track of the node color
  - we keep track of the node's parent
    - this is useful for some of the algorithms, allows us to traverse back up the tree.

Balancing and Rotations

Balancing Operations

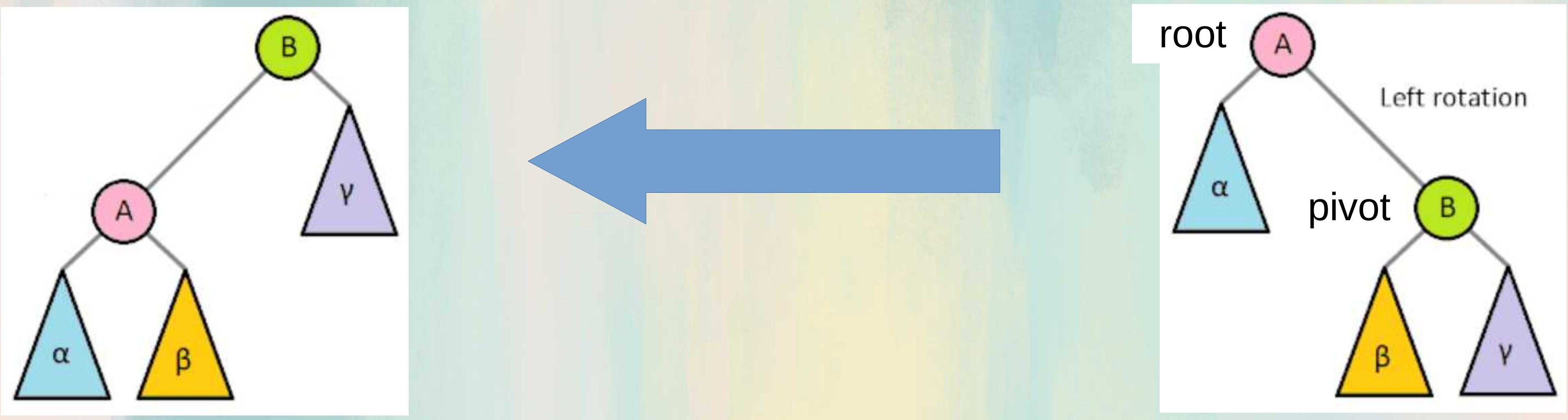
- After a node has been added or removed from the tree, the tree must be checked to make sure that all properties are maintained.
- insert() and delete() each have a clean-up operation which checks for any property violations and then adjusts the tree as necessary.
- The clean-up is performed after every insertion and deletion.

Tree Rotations

- ***tree rotation***: an operation on a binary tree that changes the structure without interfering with the inorder sequence of the elements.
  - generally will move a node (and its subtrees) down and will move another node (and its subtrees) up.
  - decreases the height by moving smaller subtrees down and larger subtrees up, maintaining a balance on both sides of any subtree.
- Rotations are part of insert() and delete() corrections.
- terminology:
  - ***root***: the root of a subtree, this is the node which moves down.
  - ***pivot***: a child of the root, this is the node that moves up to replace its parent (which is the root of the rotation).

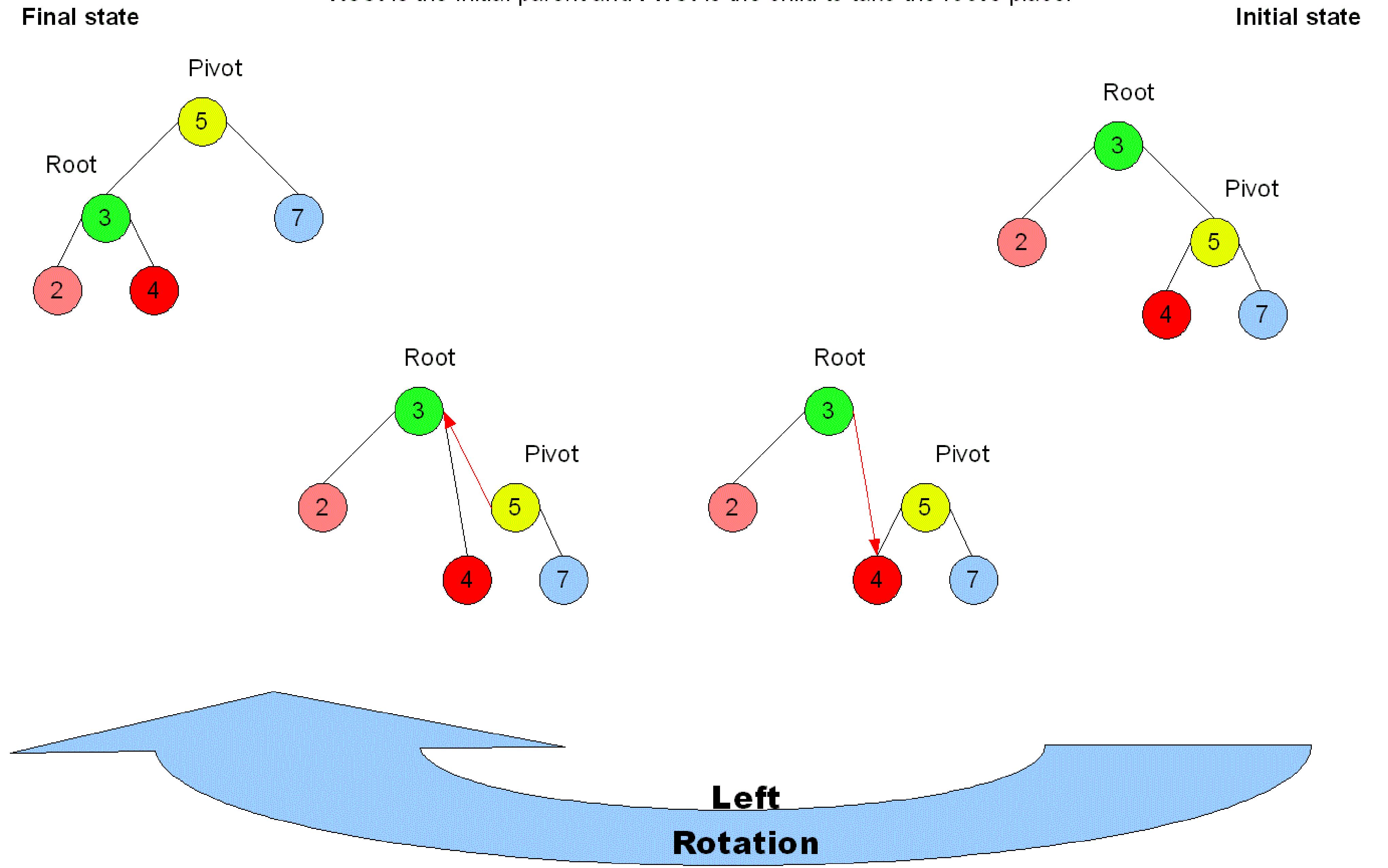
Left Rotation

- the root moves down and to the left
- the pivot moves up and to the right.
- the left subtree of the pivot becomes the right subtree of the root after the movement.



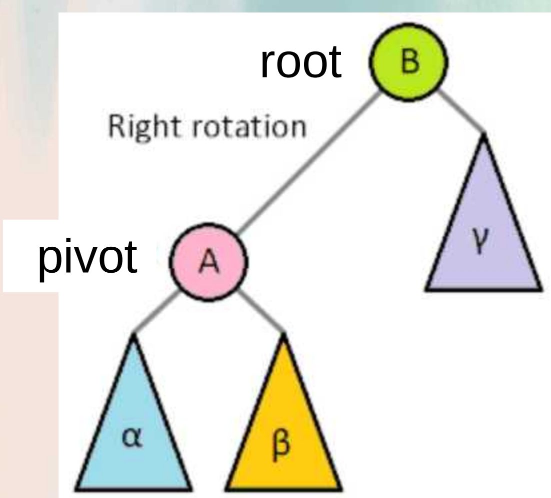
Left Rotation

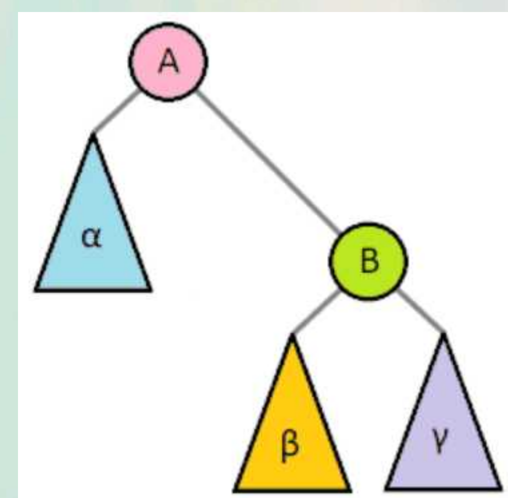
Root is the initial parent and **Pivot** is the child to take the root's place.



Right Rotation

- the root moves down and to the right
- thevpivot moves up and to the left.
- the right subtree of the pivot becomes the left subtree of the root after the movement.

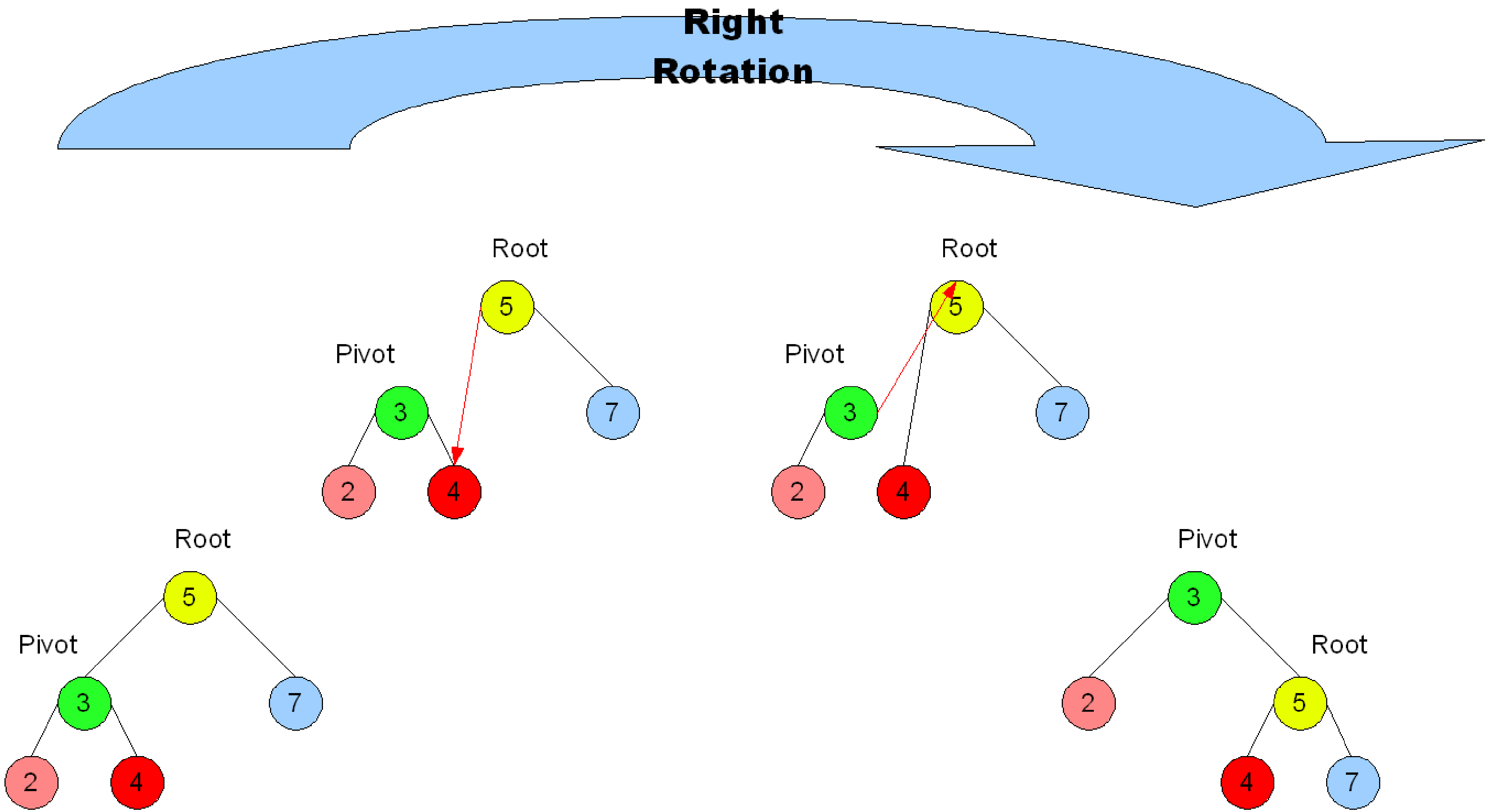




Right Rotation

Initial state | **Final state**

Root is the initial parent and **Pivot** is the child to take the root's place.



Hints about Rotation

- Be sure to update all parent references correctly.
- After the rotation, make sure that the new root of the subtree (the old pivot) correctly attaches to the node above it.
- What happens when you rotate with the root of the entire tree as the root of the rotation.?
  - Where do you attach the new root (old pivot) when the root node of a tree has no parent?

Red-Black Tree Insertion Algorithm

Notations Used in Algorithm

- N is the current node.
- P is the parent of the current node (N).
- U is the uncle of the current node (N).
  - an uncle is the sibling of a node's parent.
- G is the grandparent of the current node (N).
  - grandparent is the parent of a node's parent.

insert()

- Insert the new value using the BST insert() algorithm. (refer to last week's lecture).
- All new nodes should start out as red:
  - if the new node was black, it could potentially violate property 4.
  - we do not want to have more black nodes than necessary because of the balancing requirement of the red-black tree.
- So, we insert a new node according to BST insert() and we color it red.
- Once the insertion has been performed, we need to check our tree and "clean-up" any violations that may have been made by inserting a new node...

insertCleanup()

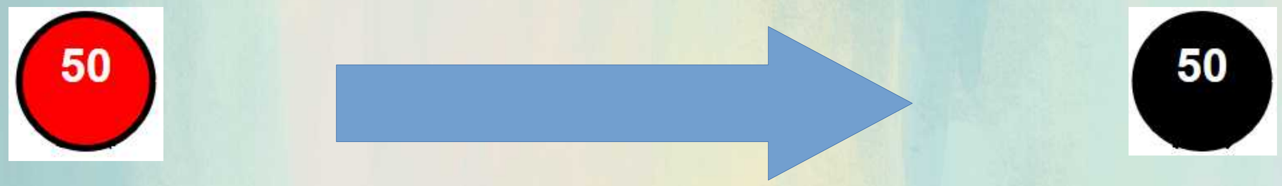
- After the new node has been placed, we need to consider the following five cases in order to make sure the tree maintains the previous properties:

- Case 1: The root node is red.
- Case 2: N's parent (P) is black.
- Case 3: N's parent (P) and uncle (U) are red.
- Case 4: N's parent (P) is red and uncle (U) is black.
  - Case 4a: N is a right child of P and P is a left child of G
  - Case 4b: N is a left child of P and P is a right child of G
- Case 5: N's parent (P) is red and uncle (U) is black.
  - Case 5a: N is a left child of P and P is a left child of G.
  - Case 5b: N is a right child of P and P is a right child of G.

- NOTE: insertionCleanup() is a recursive algorithm.

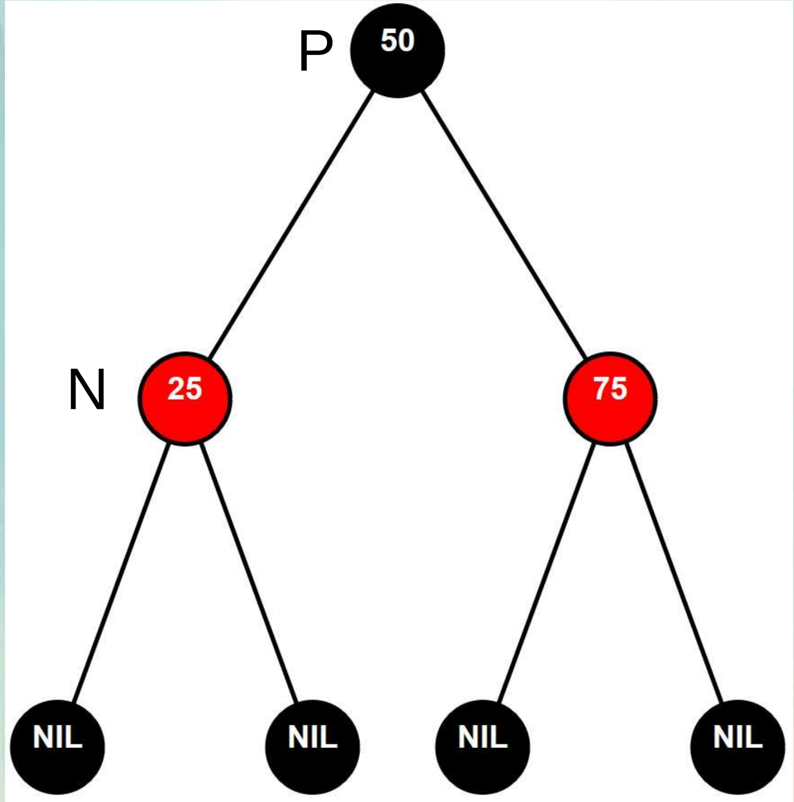
insertionCleanup() - Case 1

- **Case 1: The root node of entire tree is red.**
  - change root color from red to black
  - insertionCleanup() is finished.
- property 2 is satisfied
- property 5 is satisfied
  - setting the root to be black, adds 1 black node to all paths of the tree.



insertionCleanup() - Case 2

- **Case 2: N's parent (P) is black.**
  - No changes necessary.
  - insertionCleanup() is finished.

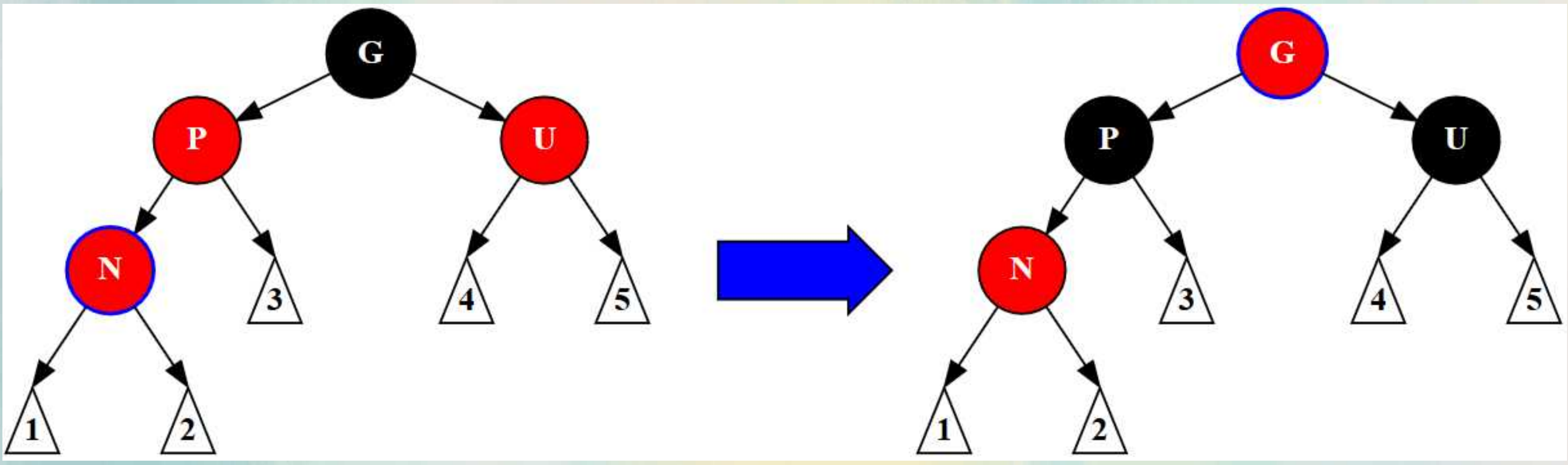


- All properties satisfied since a black parent can always have red children.

- Example: Adding 25 and 75 to the black parent cause no violations.

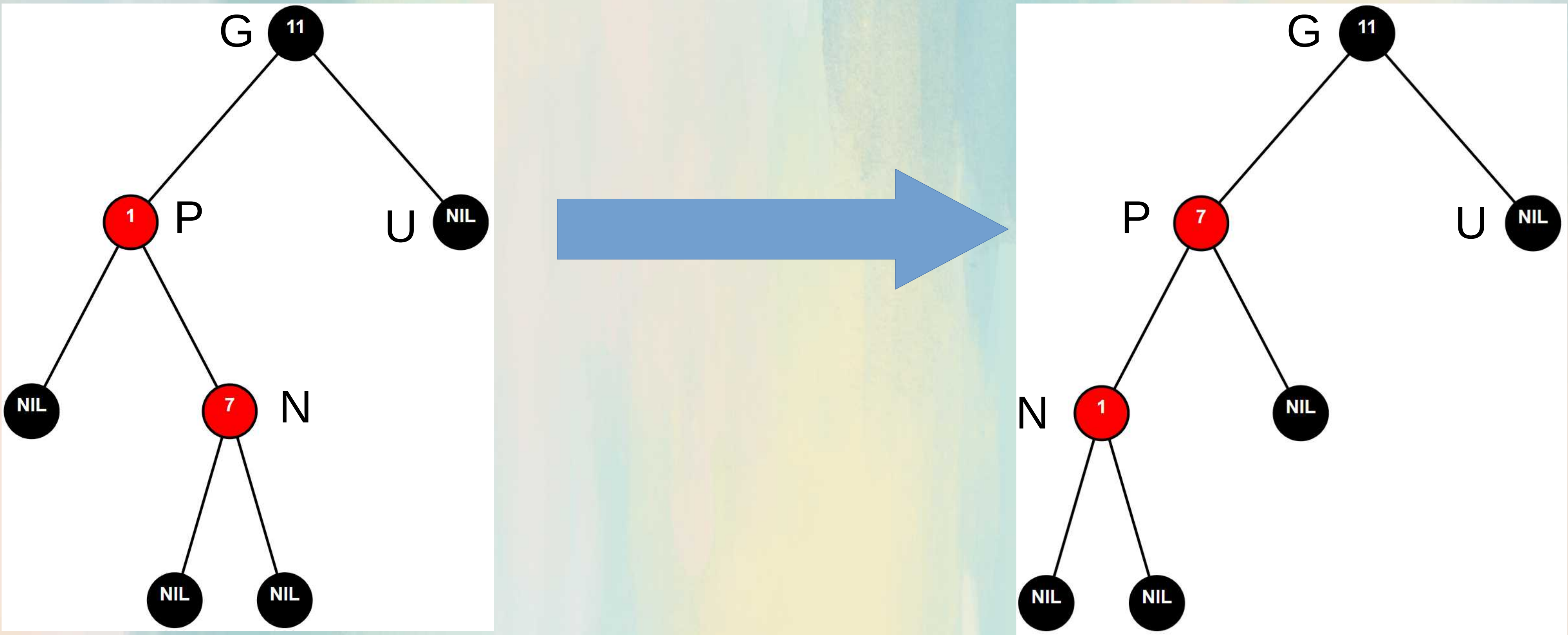
insertionCleanup() - Case 3

- **Case 3: N's parent (P) and uncle (U) are red.**
  - change color of P and U to black
  - change grandparent (G) of N to red.
  - recursively check G for any further violations (starting from case 1)
  - insertionCleanup() is finished.
- Helps to correct property 4.
- Once we make these color changes, it is possible for the grandparent (G) to violate property 2 (the root is black) or property 4 (both children of every red node are black).
- To correct this violation, we recursively check the previous cases starting from case 1, on the grandparent (G).



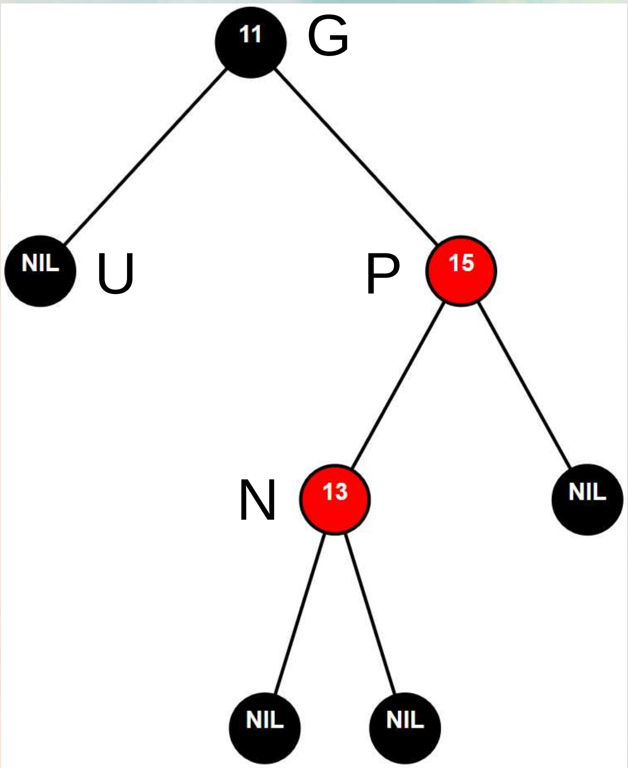
insertionCleanup() - Case 4 and 4a

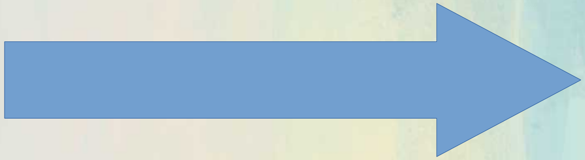
- **Case 4: N's parent (P) is red and its uncle (U) is black.**
  - *Case 4a: N is a right child of its parent (P) and P is a left child of its parent (G).*
    - left rotate the subtree rooted at P (P is the root of the rotation, and N is the pivot)
    - update N to point to P (because the parent was rotated to the left and we need to check the new N (old parent) against case 5 later on.
    - continue to case 5

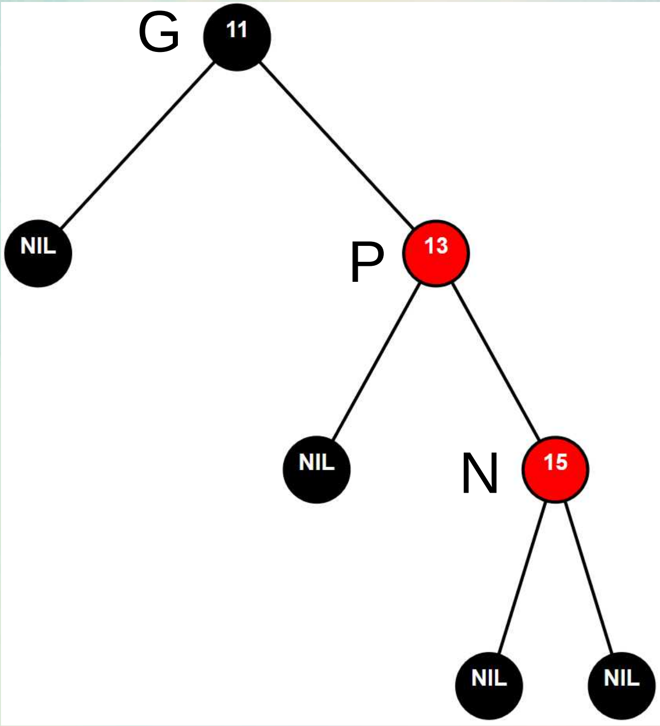


insertionCleanup() - Case 4 and 4b

- **Case 4: N's parent (P) is red and its uncle (U) is black.**
  - *Case 4b: N is a left child of its parent (P) and N's parent is a right child of the grandparent (G)*
    - right rotate the subtree rooted at P (P is the root of the rotation, and N is the pivot)
    - update N to point to P (because the parent was rotated to the right and we need to check the new N (old parent) against case 5 later on.
    - continue to case 5

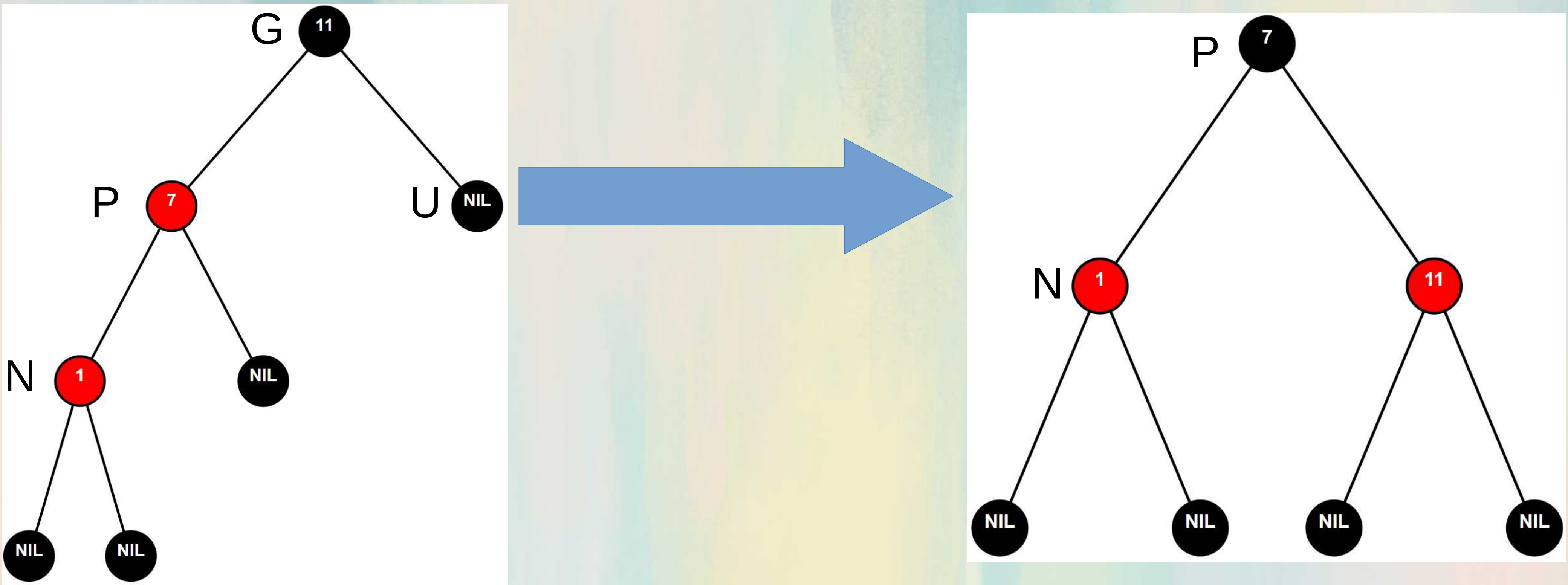






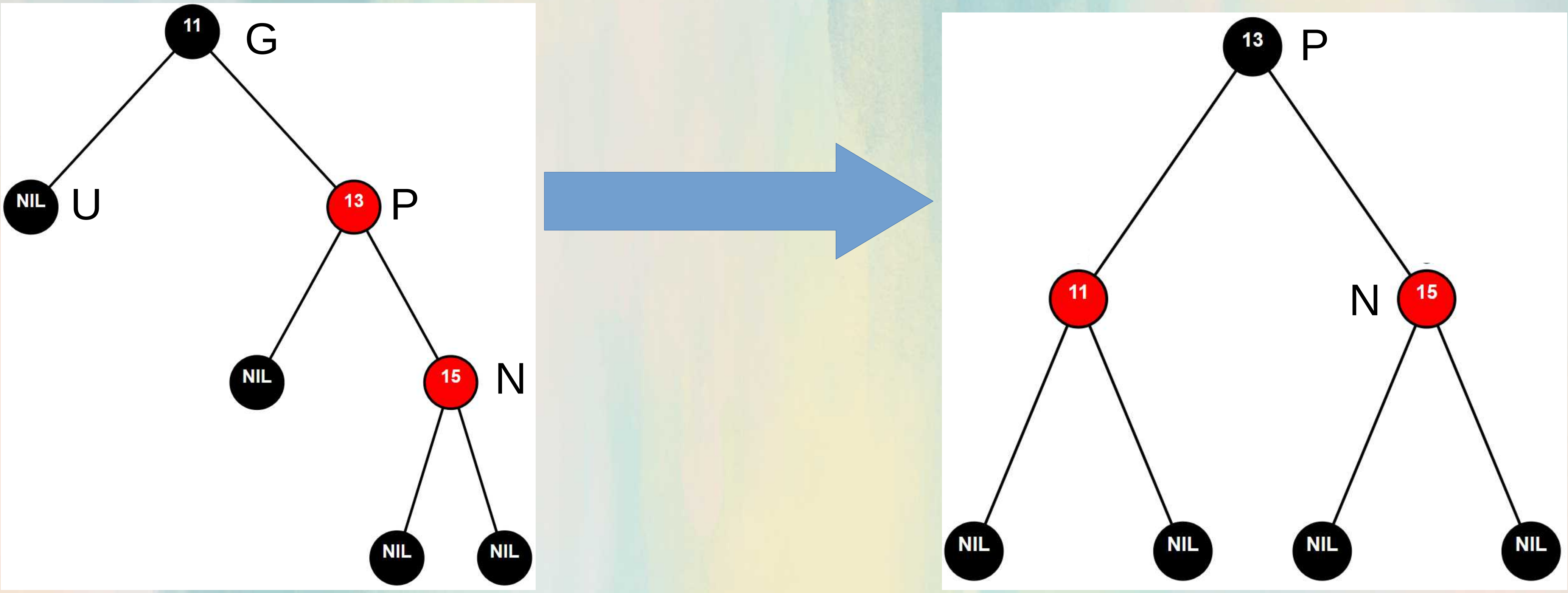
insertionCleanup() - Case 5 and 5a

- **Case 5: N's parent (P) is red and its uncle (U) is black.**
  - *Case 5a: N is a left child of its parent (P) and P is a left child of its parent (G).*
    - Change the color of P to black.
    - Change the color of G to red
    - right rotate using G as the root of the rotation
    - insertionCleanup() is finished.



insertionCleanup() - Case 5 and 5b

- **Case 5: N's parent (P) is red and its uncle (U) is black.**
  - *Case 5b: N is a right child of its parent (P) and P is a right child of its parent (G).*
    - Change the color of P to black.
    - Change the color of G to red
    - left rotate using G as the root
    - insertionCleanup() is finished.



Red-Black Tree Delete Algorithm

delete()

- Use the delete() algorithm of a normal binary search tree with some alterations (see next slide).

- In Cases 1 and 2 of delete().
  - If D is a leaf, then its child will be a NIL leaf. Recall that NIL leaves are considered to be black.
  - If D is not a leaf, then it will have one child.
  - In either case, if D or its child are red, replace D with its child and color the child black.
  - If D and its child are both black, replace D with its child and color the child "double black"

- After deleting it is possible that we end up with a "double black" node.
  - double black nodes count as two black nodes which unbalances the tree on that branch (all paths must have the same number of black nodes.)

```
delete(key):
   delete(nodeToDelete(key))

delete(node):
   if node is leaf:                    //CASE 1
     if isLeftChild(node):
       node.parent.left = NIL
     else if isRightChild(node):
       node.parent.right = NIL

     //We must set the parent of NIL to be node's
     //parent for the fixDoubleBlack() method.
     NIL.parent = node.parent

     if node.color is black:
       NIL.color = double black
       fixDoubleBlack(NIL)

     //delete continued on next slide...
```

```
//continued from previous slide

else if numChildren(node) == 1: //Case 2
  child = get the left or right child of node
  if isLeftChild(node):
     node.parent.left  = child
  else if isRightChild(node):
     node.parent.right = child

  if child.color or node.color is red:
     child.color = black
  else if child.color and node.color is black:
     child.color = double black
     fixDoubleBlack(child)

else if numChildren(node) == 2:  //Case 3
  max = maxLeftSubtree(node)
  node.setItem(max.getItem())
  delete(max)
```

Notations Used in Algorithm

- N is the current node.
  - N is also the double black node.
- P is the parent of the current node (N).
- S is the sibling of the current node (N).
- RC is the red child of the sibling of (N).
  - Also could be called the nephew or niece of N.

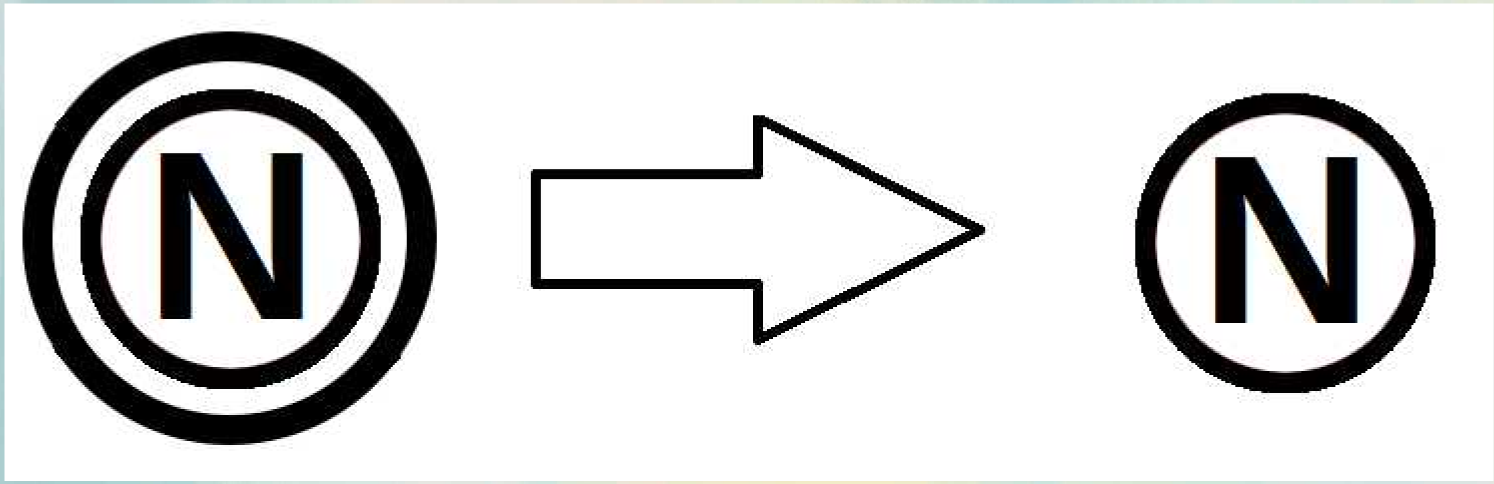
fixDoubleBlack()

- Once you have a double black node in the tree it is important to fix that node and get rid of the double black color.
- fixDoubleBlack() will use a series of rotation or recoloring operations to correct the imbalance in color.

fixDoubleBlack() - Case 1

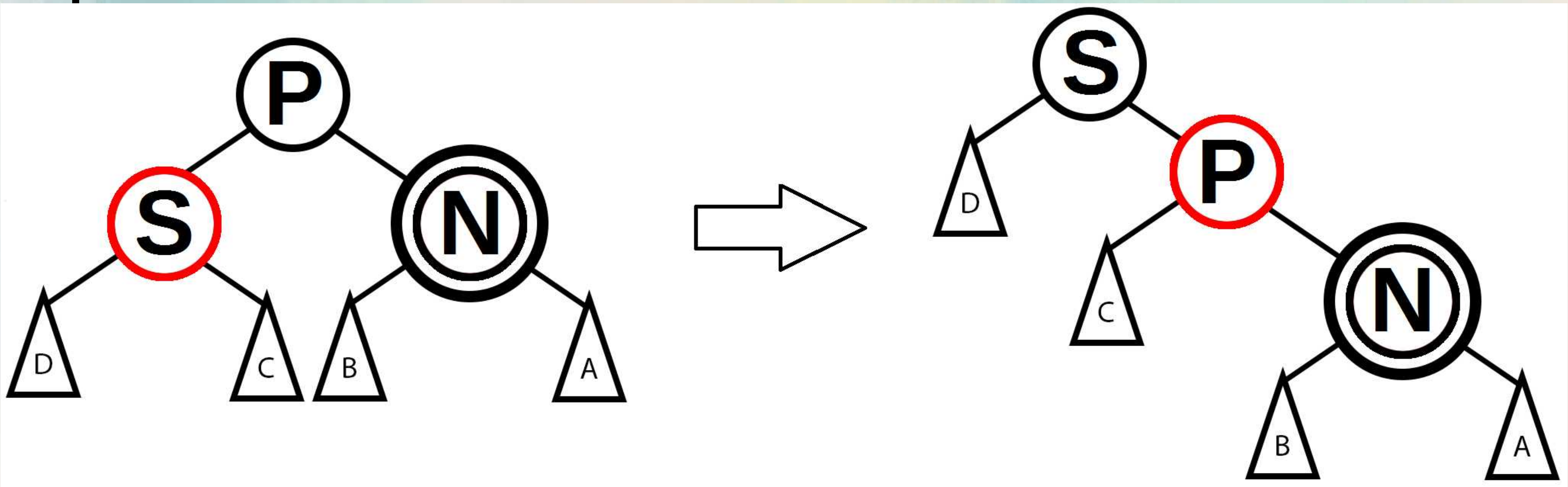
- **Case 1: N is the root of the entire tree.**
  - Change color of N to black.
  - fixDoubleBlack() is finished.

- Simple base case, we can remove a black color from the root which reduces the number of black nodes on all paths by 1.



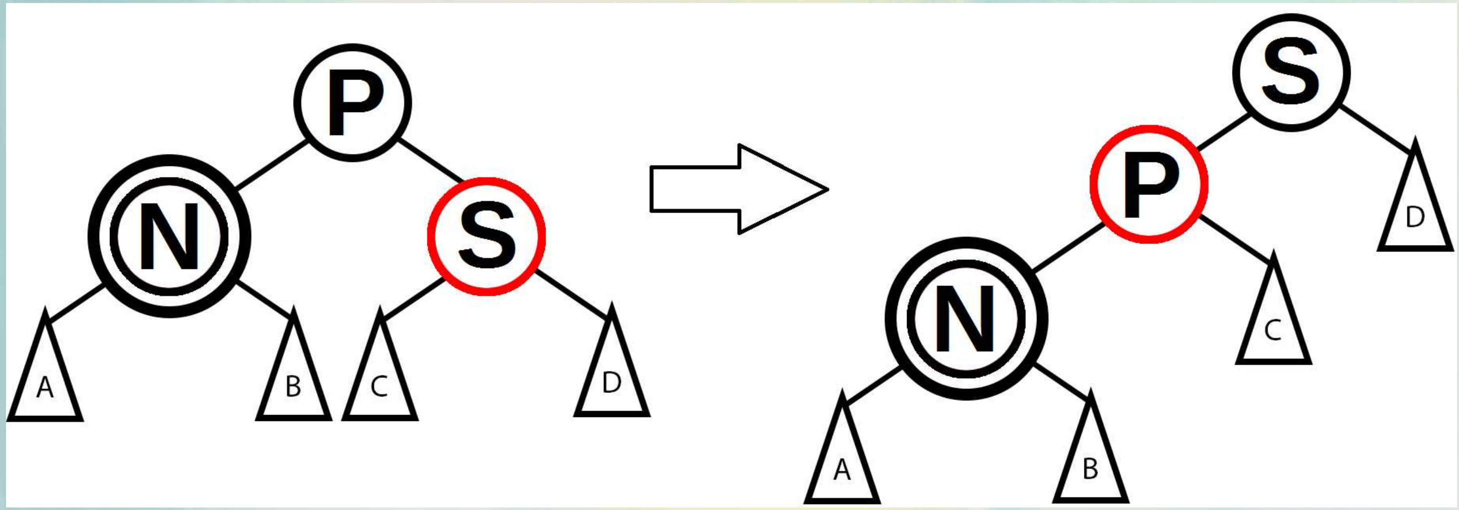
fixDoubleBlack() - Case 2 and 2a

- **Case 2: Sibling (S) of N is red:**
  - *Case 2a: N is a right child of its parent P:*
    - change color of S to black
    - change color of P to red
    - right rotate with P as the root of rotation
    - recurse with N fixDoubleBlack(N).
- At this point we still have not fixed the double black, so we need to check all cases again with N in its new position.



fixDoubleBlack() - Case 2 and 2b

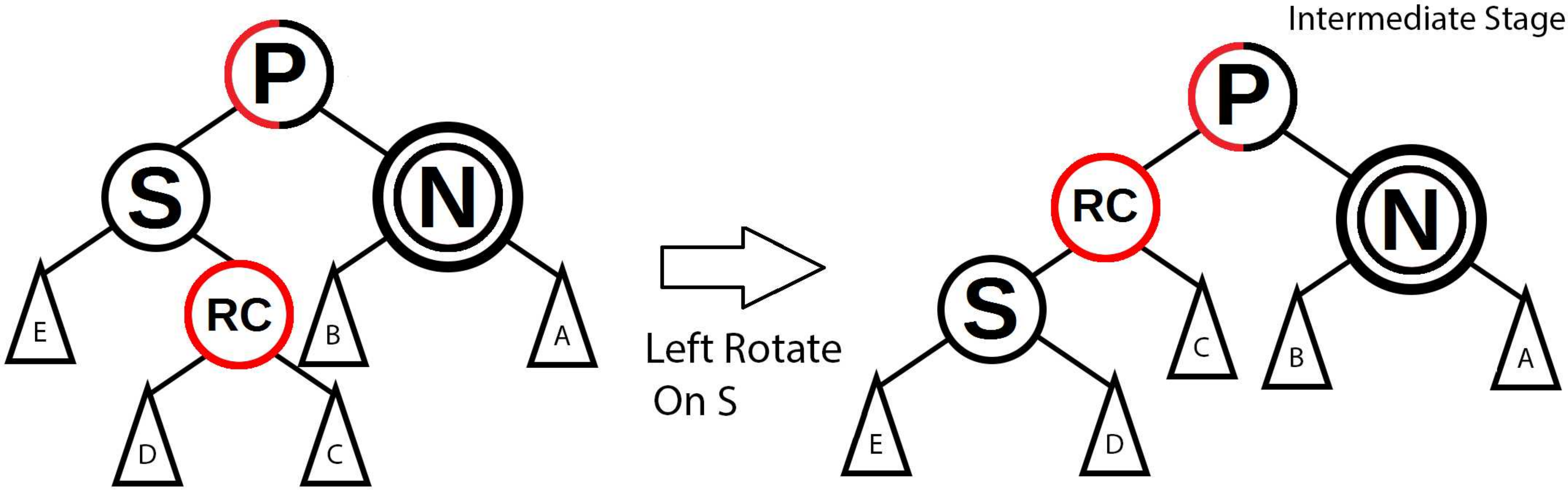
- **Case 2: Sibling (S) of N is red:**
  - *Case 2b: N is a left child of its parent P:*
    - change color of S to black
    - change color of P to red
    - left rotate with P as the root of rotation
    - recurse with N fixDoubleBlack(N).
- At this point we still have not fixed the double black, so we need to check all cases again with N in its new position.

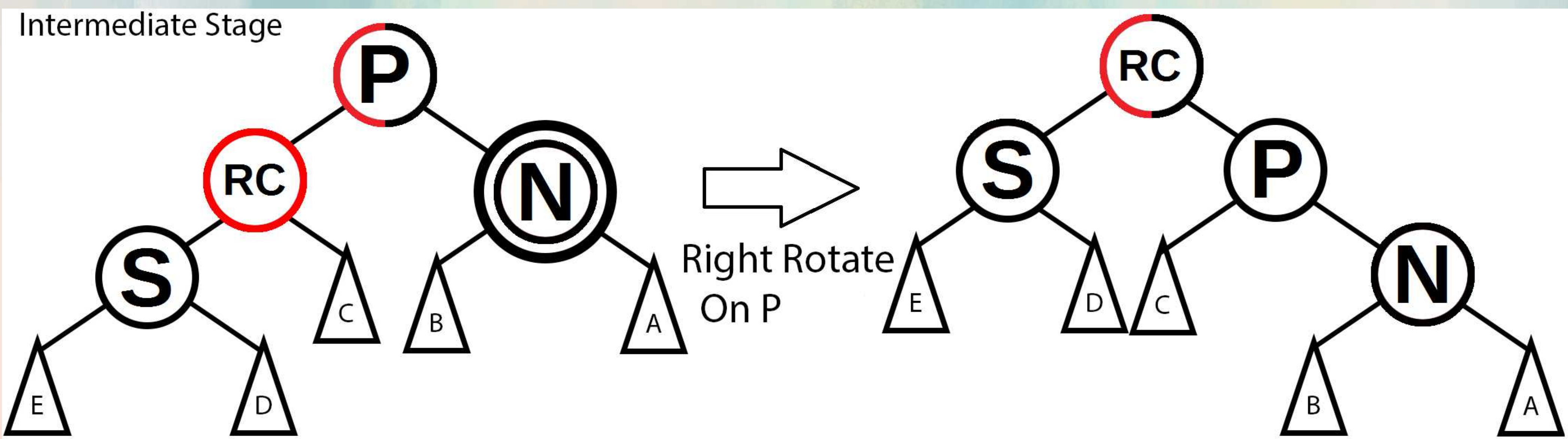


fixDoubleBlack() - Case 3, 3a, and 3a.1

- **Case 3: Sibling (S) of N has at least one red child (RC)**
  - (NOTE: If S has two red children, it does not matter if you use Case 3a.1 or 3a.2, choose one.)
  - *Case 3a: S is the left child of its parent P*
    - *Case 3a.1: RC is the right child of S*
      - left rotate using S as the root of rotation.
      - right rotate using P as the root of rotation.
      - change color of RC to be the color of P
      - change color of S to black.
      - change color of P to black.
      - change color of N to black.
      - fixDoubleBlack() is finished.

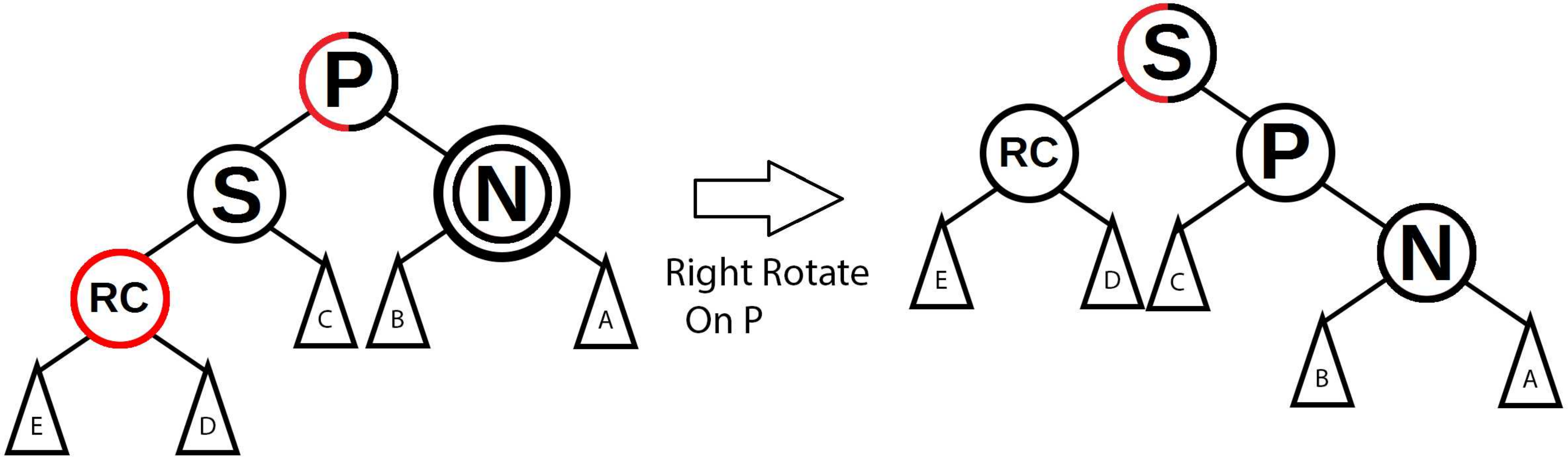
- See next slide for diagrams. NOTE: The half red half black color of a node means it could be either red or black and that depends on the algorithm.


P
S
N
RC
E
B
A
D
C
Left Rotate
On S
Intermediate Stage
P
RC
N
S
C
B
A
E
D


Intermediate Stage
P
RC
N
S
C
B
A
E
D
Right Rotate
On P
RC
S
P
N
E
D
C
B
A

fixDoubleBlack() - Case 3, 3a, and 3a.2

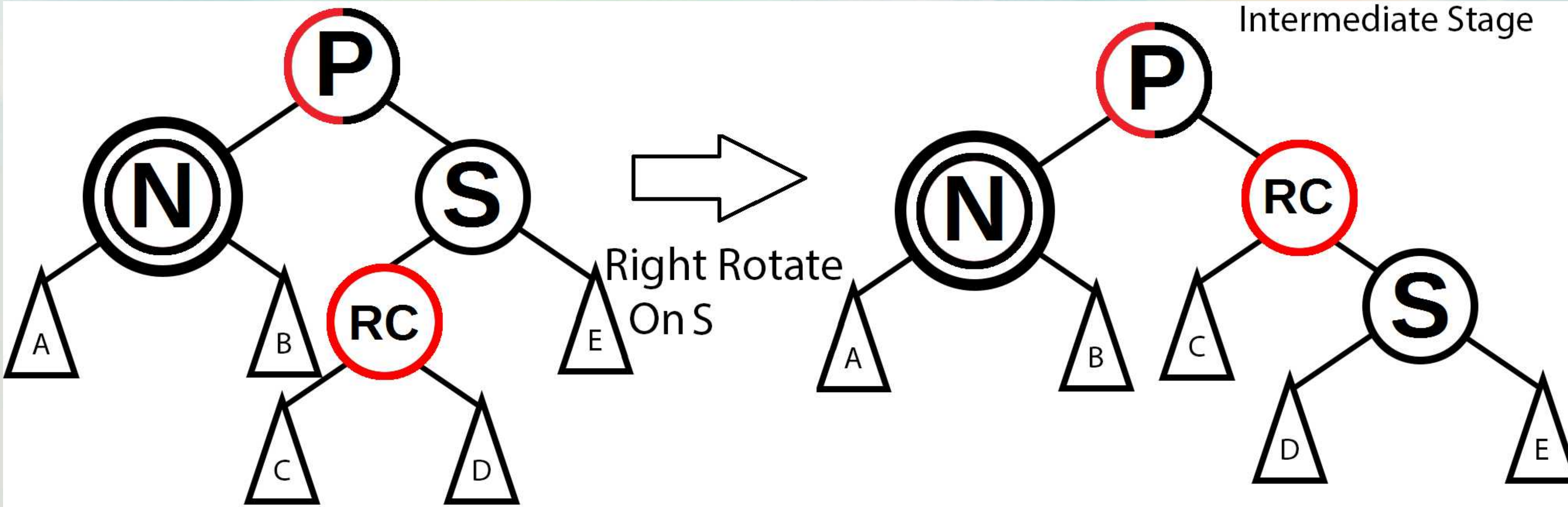
- **Case 3: Sibling (S) of N has at least one red child (RC)**
  - (NOTE: If S has two red children, it does not matter if you use Case 3a.1 or 3a.2, choose one.)
  - *Case 3a: S is the left child of its parent P*
    - *Case 3a.2: RC is the left child of S*
      - right rotate using P as the root of rotation.
      - change color of S to be the color of P
      - change color of RC to black.
      - change color of P to black.
      - change color of N to black.
      - fixDoubleBlack() is finished.
- See next slide for diagrams. NOTE: The half red half black color of a node means it could be either red or black and that depends on the algorithm.

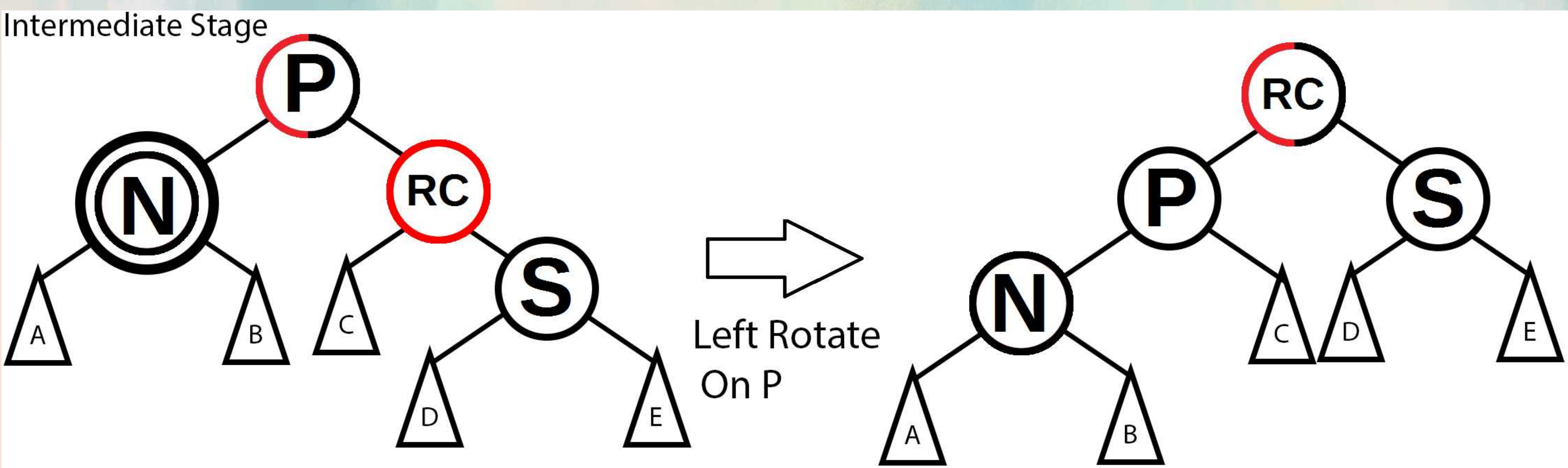


P
S
N
RC
E
D
C
B
A
Right Rotate
On P
S
RC
P
E
D
C
N
B
A

fixDoubleBlack() - Case 3, 3b, and 3b.1

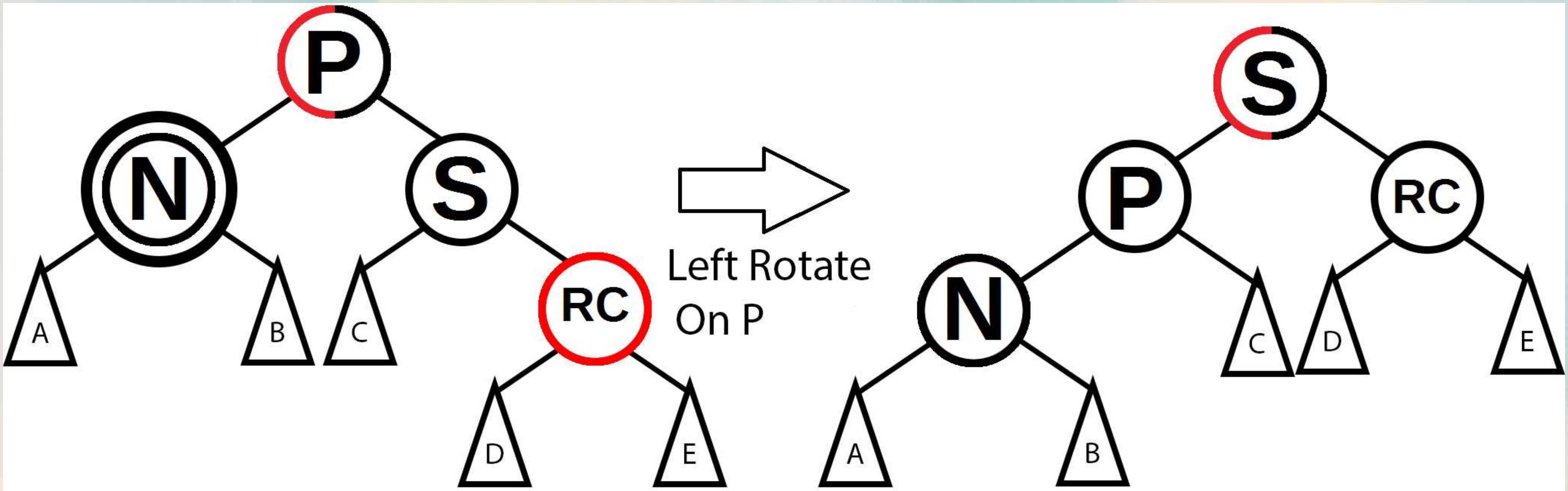
- **Case 3: Sibling (S) of N has at least one red child (RC)**
  - (NOTE: If S has two red children, it does not matter if you use Case 3b.1 or 3b.2, choose one.)
  - *Case 3b: S is the right child of its parent P*
    - *Case 3b.1: RC is the left child of S*
      - right rotate using S as the root of rotation.
      - left rotate using P as the root of rotation.
      - change color of RC to be the color of P
      - change color of S to black.
      - change color of P to black.
      - change color of N to black.
      - fixDoubleBlack() is finished.
- See next slide for diagrams. NOTE: The half red half black color of a node means it could be either red or black and that depends on the algorithm.


Intermediate Stage
P
N
S
RC
A
B
C
D
E
Right Rotate
On S


Intermediate Stage
P
N
RC
S
A
B
C
D
E
Left Rotate
On P

fixDoubleBlack() - Case 3, 3a, and 3a.2

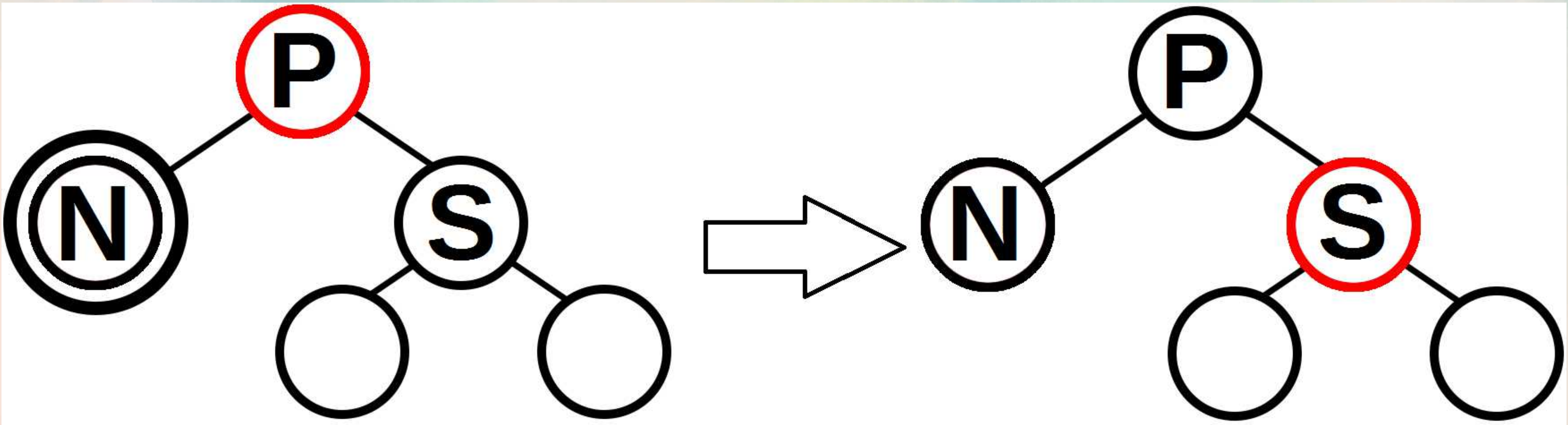
- **Case 3: Sibling (S) of N has at least one red child (RC)**
  - (NOTE: If S has two red children, it does not matter if you use Case 3b.1 or 3b.2, choose one.)
  - *Case 3b: S is the right child of its parent P*
    - *Case 3b.2: RC is the right child of S*
      - left rotate using P as the root of rotation.
      - change color of S to be the color of P
      - change color of RC to black.
      - change color of P to black.
      - change color of N to black.
      - fixDoubleBlack() is finished.
- See next slide for diagrams. NOTE: The half red half black color of a node means it could be either red or black and that depends on the algorithm.



P
N
S
A
B
C
RC
D
E
Left Rotate
On P
S
P
RC
N
C
D
E
A
B

fixDoubleBlack() - Case 4 and 4a

- **Case 4: Sibling (S) and both of its children are black.**
  - *Case 4a: Parent (P) of S is red:*
    - change S to red
    - change P to black
    - change N to black
    - fixDoubleBlack() is finished
- NOTE: For this case, the side that N or S is on does not matter.



fixDoubleBlack() - Case 4 and 4b

- **Case 4: Sibling (S) and both of its children are black.**
  - *Case 4b: Parent (P) of S is black:*
    - change S to red
    - change P to double black
    - change N to black
    - recurse, fixDoubleBlack(P)

- NOTE: For this case, the side that N or S is on does not matter.