

# **Data Structures:**

## **Height-Balanced Search Trees:**

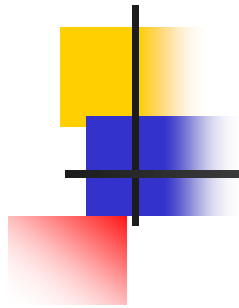
### **Red-Black Trees**

---

**YoungWoon Cha**

**(Slide credits to Won Kim)**

**Spring 2022**



# Red-Black Tree



# Red-Black Tree

---

- Height balanced binary search tree (not perfectly, similar to the AVL Tree)
- $O(\log_2 n)$  avg and worst-case performance for search, insert, and delete
- Invented in 1972 by Rudolf Bayer and named symmetric binary B-tree
- Renamed Red-Black Tree in 1978 by Leonidas J. Guibas and Robert Sedgwick



# Applications

---

- Time-sensitive applications such as real-time applications
- Building block in other data structures which provide worst-case guarantees; for example, many data structures used in computational geometry
- The Completely Fair Scheduler used in current Linux kernels

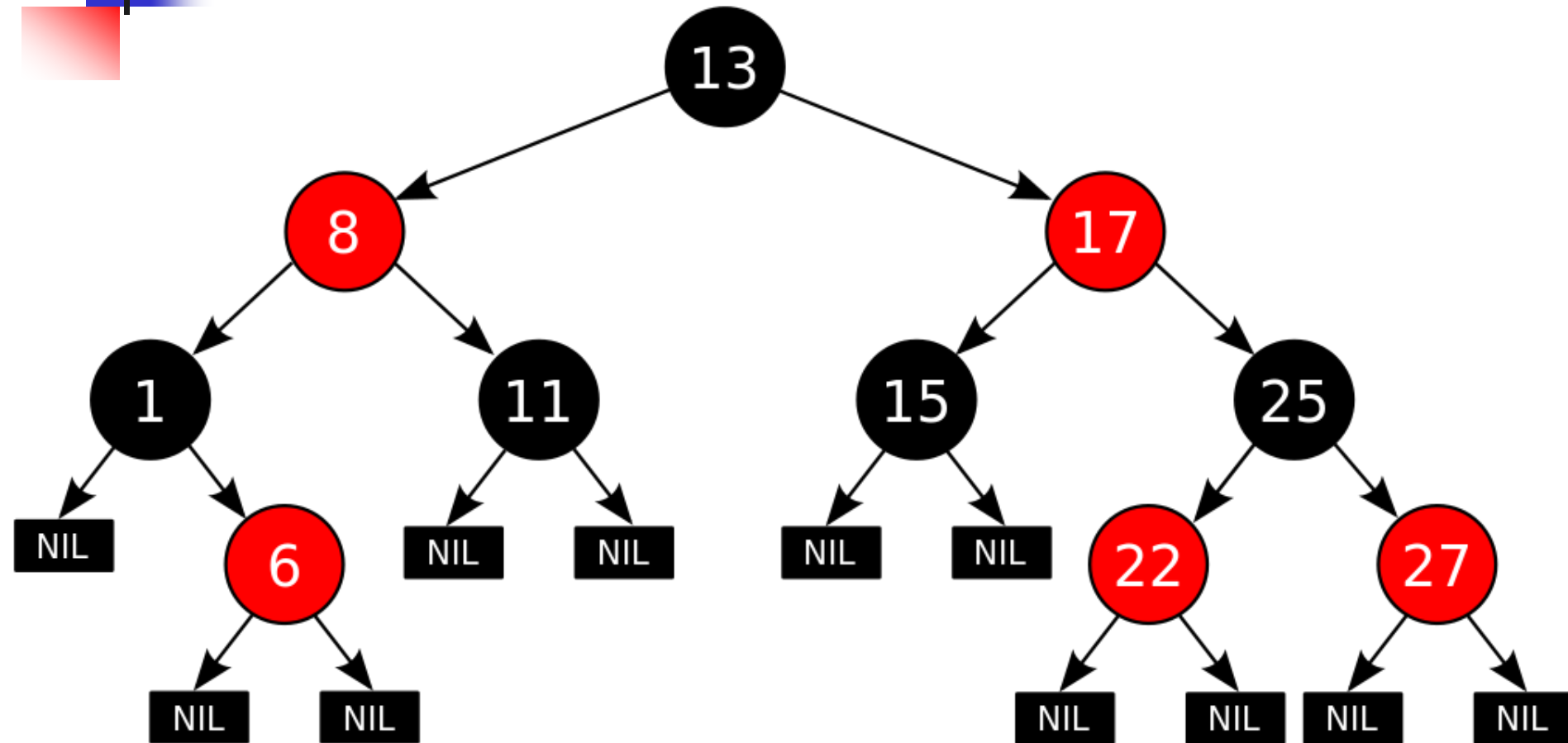


# Primary Characteristics

---

- Uses standard binary search tree algorithms for search, insert, and delete
- Each node is assigned a color (either **red** or **black** – hence the name of the tree)
- Height balance is done by rotations (similar to the AVL Tree) and/or changing the colors of the nodes
- In insertion and deletion, preserves five properties (**shown next**)

# Example Red-Black Tree



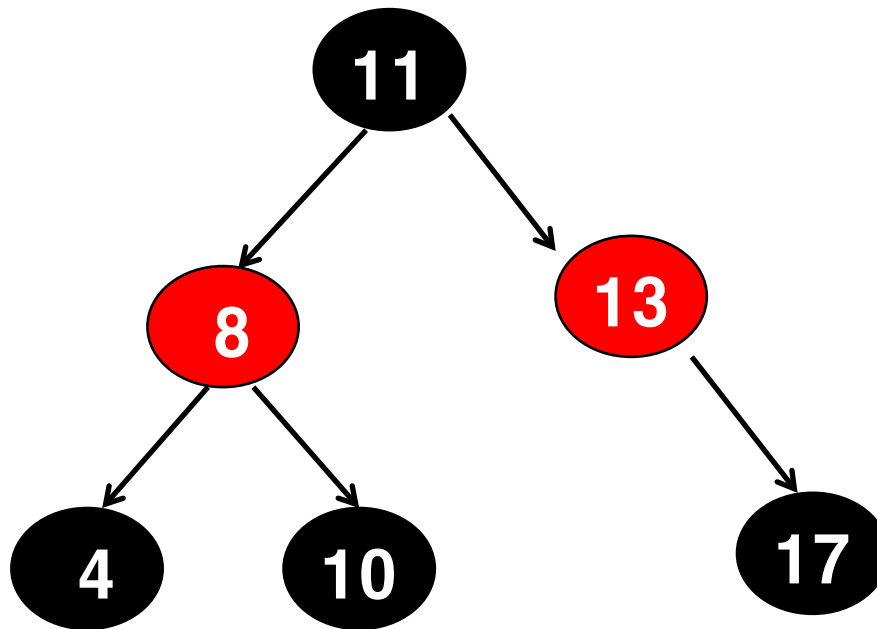


# Properties (Rules)

---

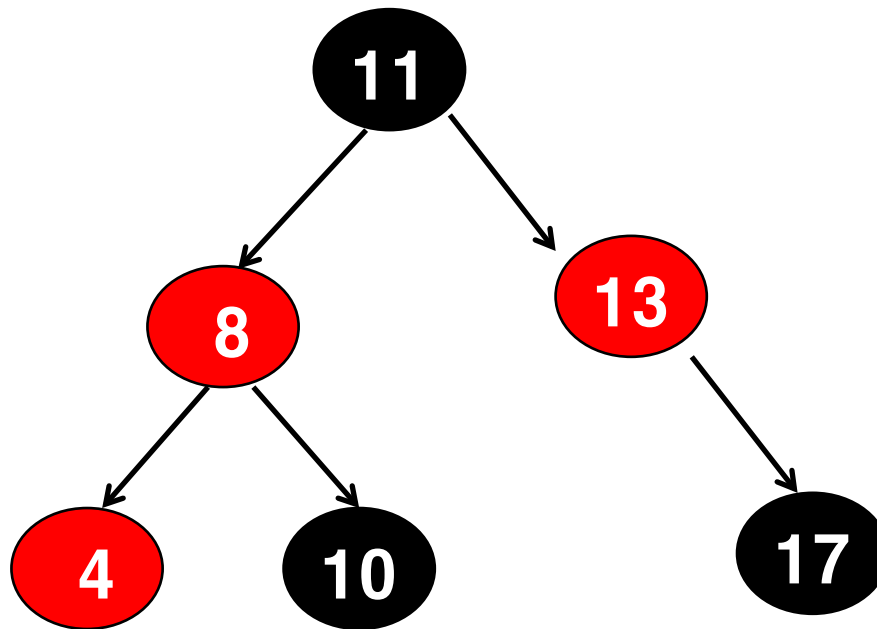
- (1) Red/Black property
  - A node is either red or black.
- (2) Root property
  - The root node is black.
- (3) External property (Leaf property)
  - All leaves (NIL) are black. *(Often omitted for convenience)*
- (4) Internal property (Red property)
  - Every red node must have two black child nodes. => **No double reds!**
- (5) Depth property
  - Every path from a given node to any of its descendant leaves contains the same number of black nodes.

# Valid Red-Black Tree? (no)

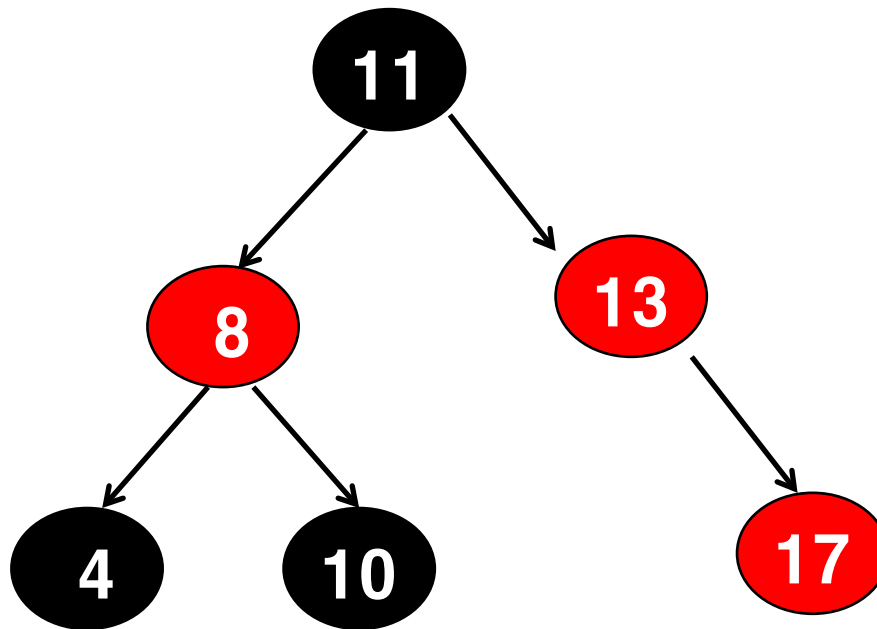




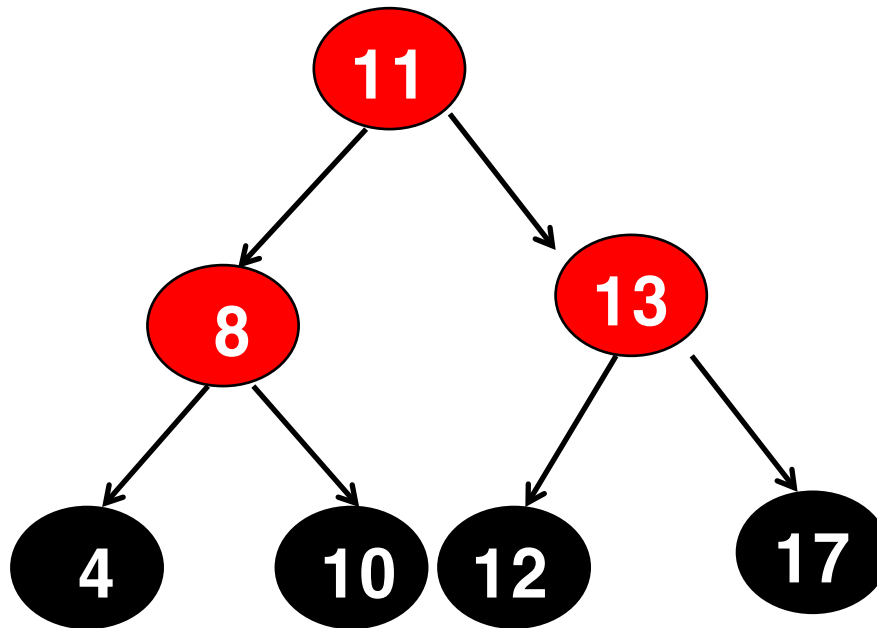
# Valid Red-Black Tree? (no)



# Valid Red-Black Tree? (no)



# Valid Red-Black Tree? (no)





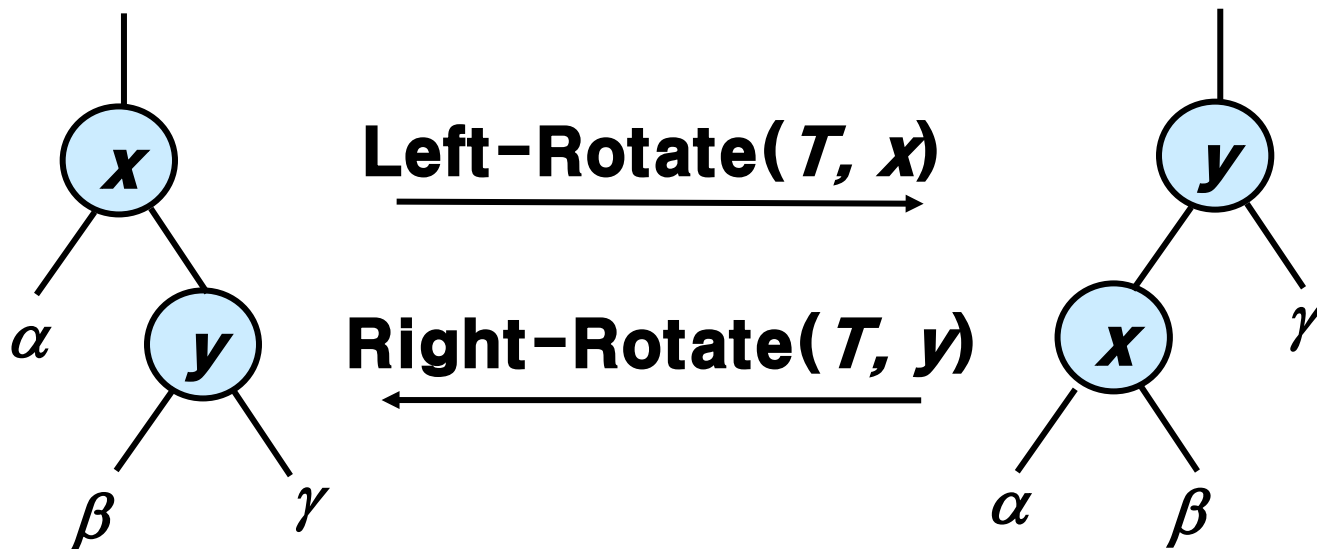
# Implementation

---

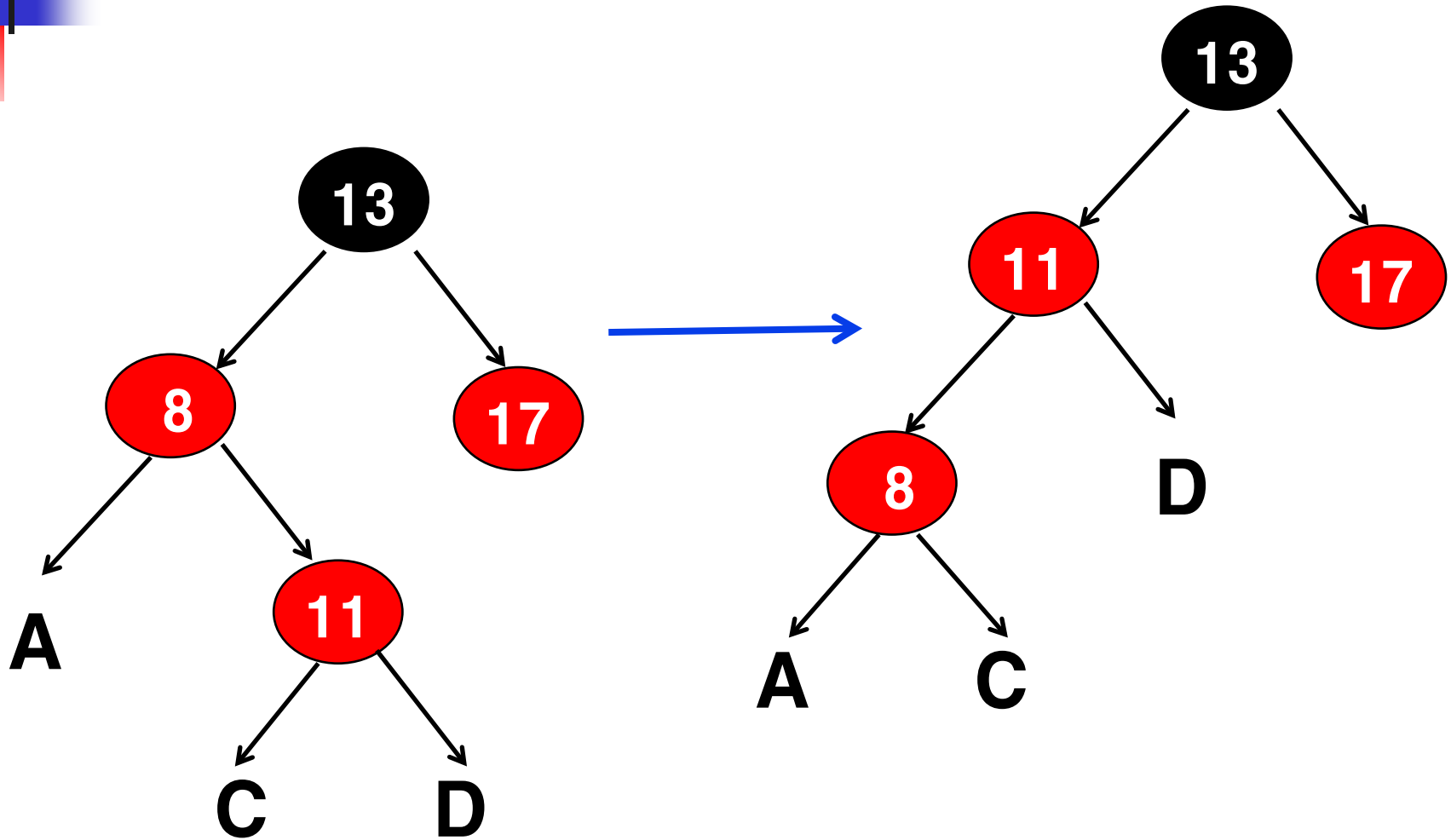
- Binary search tree + 1 bit per node: the attribute *color*, which is either **red** or **black**.
- All other attributes of BSTs are inherited:
  - *key*, *left*, *right*, and *p*.
- All empty trees (leaves) are colored black.
  - We use a single sentinel, *nil*, for all the leaves of red-black tree  $T$ , with  $color[nil] = \text{black}$ .
  - The root's parent is also  $nil[T]$ .

# Rotations

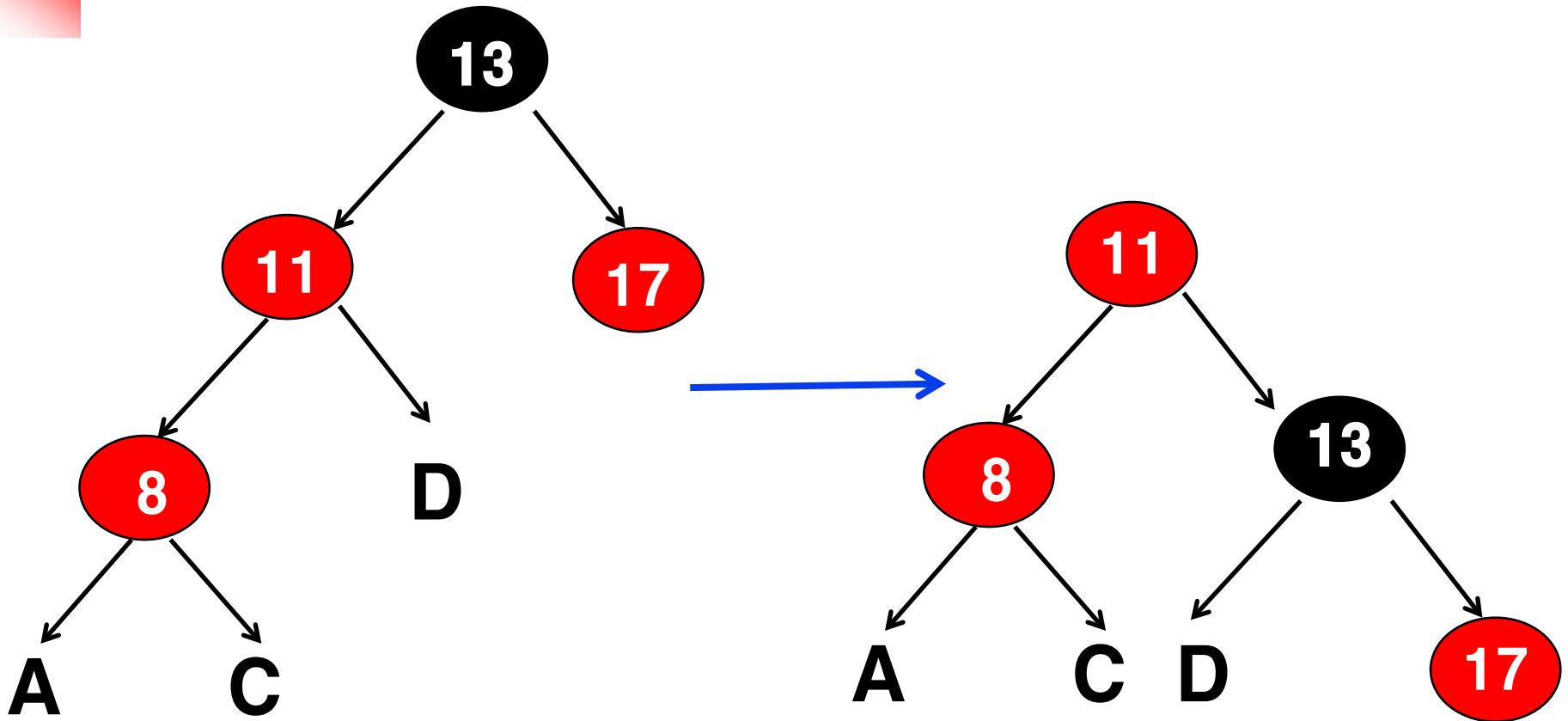
- Rotations are the basic **tree-restructuring** operation for almost all *balanced* search trees.
- Rotation takes a red-black-tree and a node,
- Changes pointers to change the local structure, and preserve the binary search tree property.
- Left rotation and right rotation are inverses.



# Left Rotation (at node 8): new node 11



## Right Rotation (at node 13): new node 11





# Insertion

---

- Add a new node as in any binary search tree insertion, and color the node **red**.
- The new red node has two black NIL nodes.
- The main property that may be violated is **two consecutive red nodes**. (red property)
- To determine which case to apply to rebalance the tree, **check the color of the uncle node** to decide the appropriate case





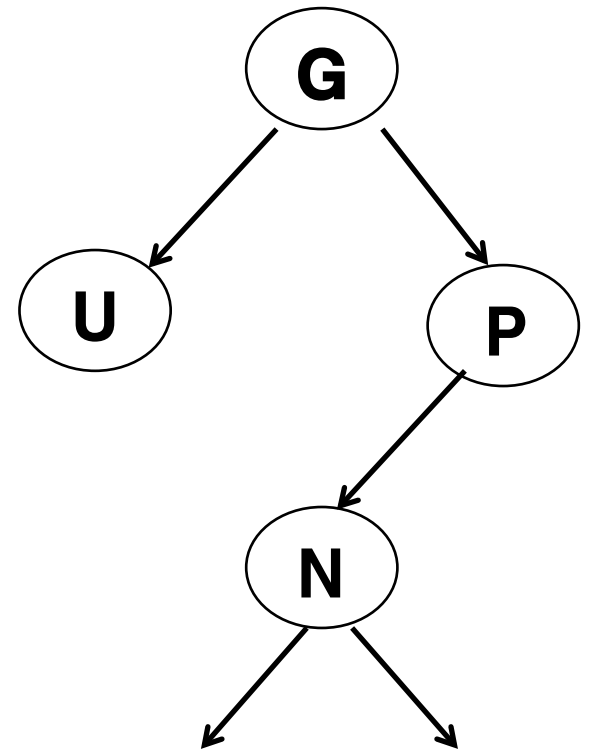
# Notes on When the Red-Black Tree Need to Be Rebalanced

---

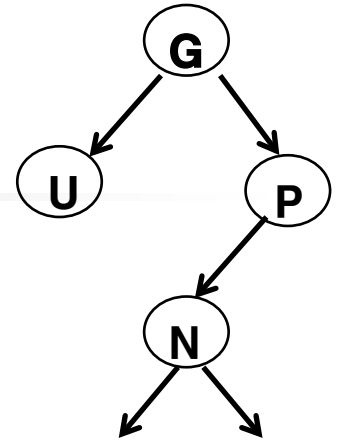
- Leaf property (all leaves are black) always holds.
- Red property (both children of every red node are black) is violated only by adding a red node
  - Recoloring: repainting a black node red
  - Restructuring: or a rotation.
- Depth property (all paths from any given node to its leaf nodes contain the same number of black nodes) is violated only by adding a black node
  - Repainting a red node black (or vice versa), or a rotation.

# Notation

- **N**: the current (new) node (colored red).
- **P**: **N**'s parent node
- **G**: **N**'s grandparent
- **U**: **N**'s uncle (P's sibling)



## Insertion: 5 Cases



- (1) **N** is the root node
- (2) **N**'s parent (**P**) is black
- (3) **N**'s parent (**P**) and uncle (**U**) are red
- (4) **N** is added as the right child of the left child of grandparent, or **N** is added as the left child of the right child of grandparent (**P** is red and **U** is black)
- (5) **N** is added as the left child of the left child of grandparent, or **N** is added as the right child of the right child of grandparent (**P** is red and **U** is black)



## Insertion: Case 1

---

- The current (new) node **N** is the root of the tree.
- N is repainted black to satisfy Root property.
- Depth property is not violated: The insertion adds one black node to every path at once.

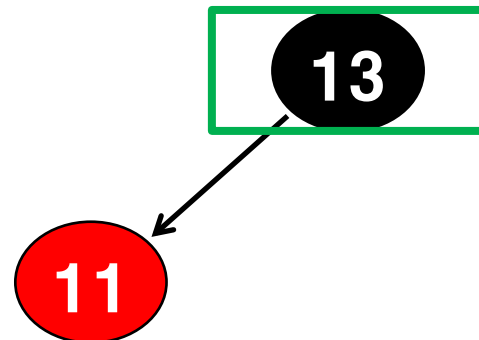
Insert 13



## Insertion: Case 2

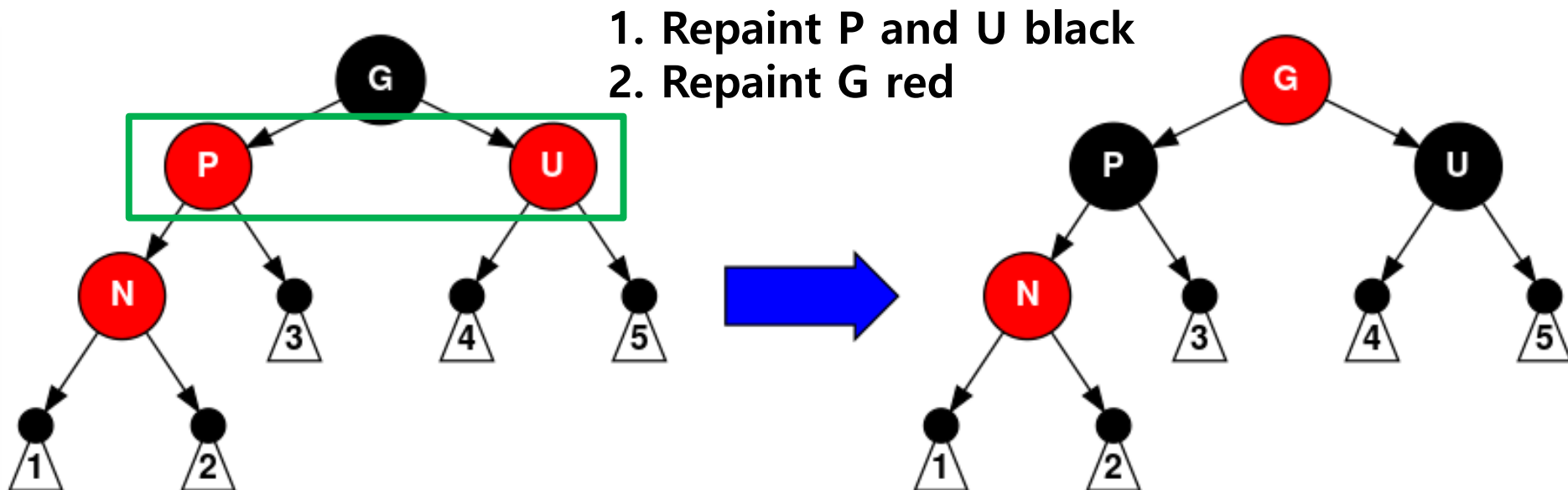
- The current node's parent **P** is black.
- The tree is still valid. Red property is not violated.
- Depth property is not violated, because the current node **N** has two black leaf children.
  - Because **N** is red, the paths through each of its children have the same number of black nodes as the path through the leaf it replaced, which was black, and so this property remains satisfied.

Insert 11



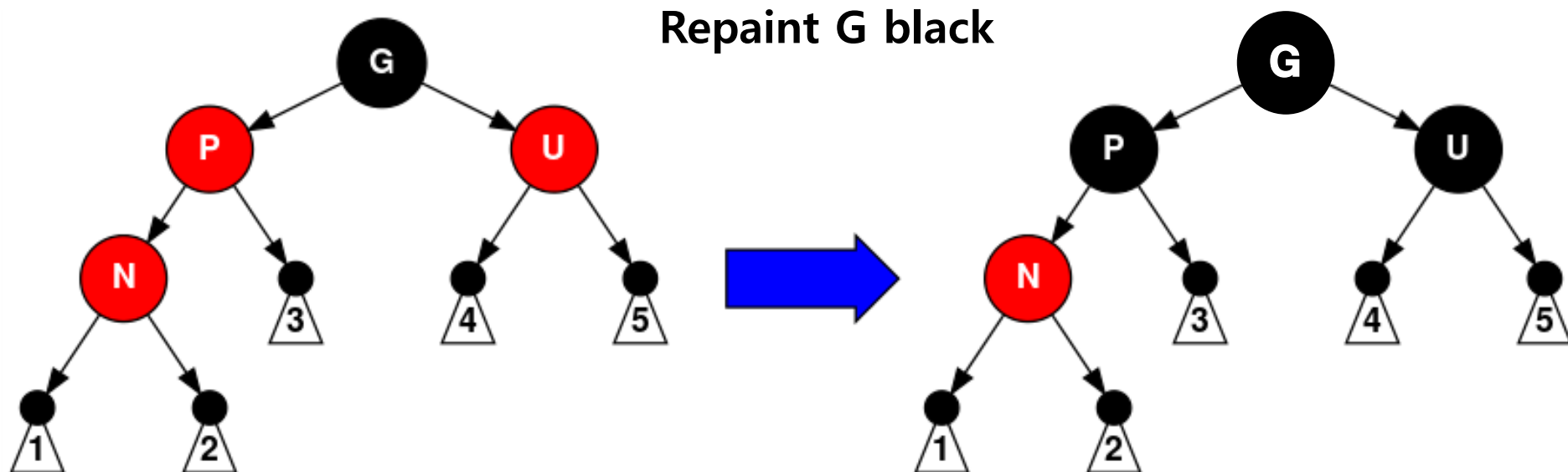
## Insertion: Case 3

- If both the parent **P** and the uncle **U** are red, both can be repainted black and the grandparent **G** becomes red (to maintain Depth property).
- Depth Property is not violated.
  - The current red node **N** has a black parent. Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed.



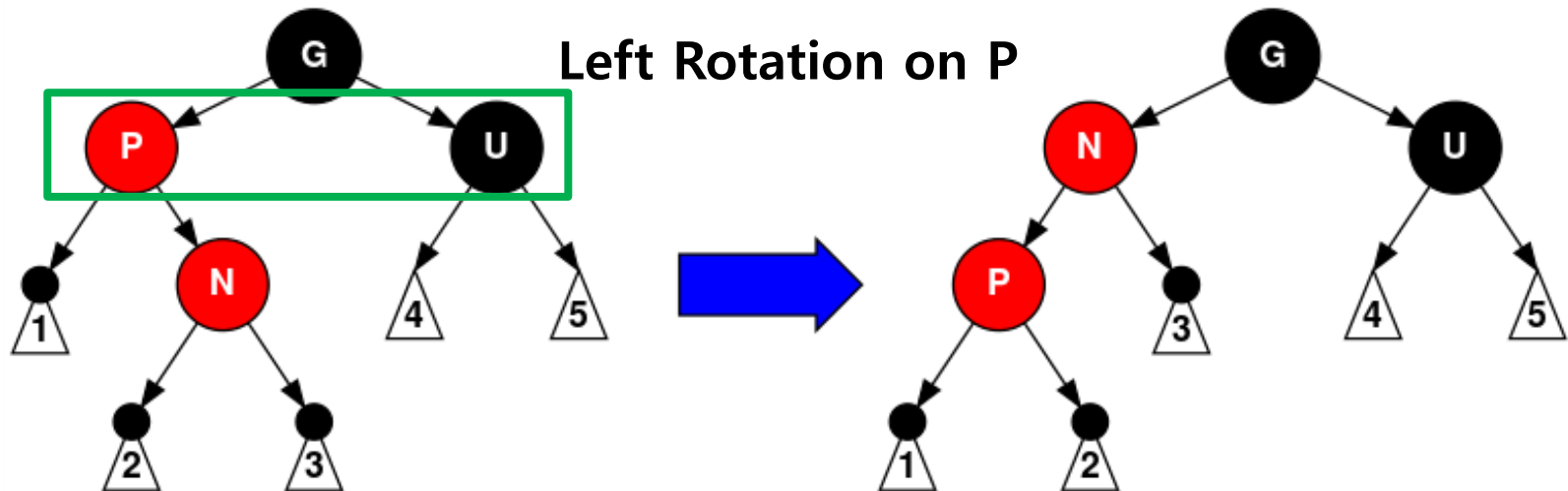
## Insertion: Case 3

- However, the grandparent **G** may now violate Root property or Red property (possibly being violated since **G** may have a red parent).
- To fix this, the entire procedure is recursively performed on **G** from case 1.



## Insertion: Case 4

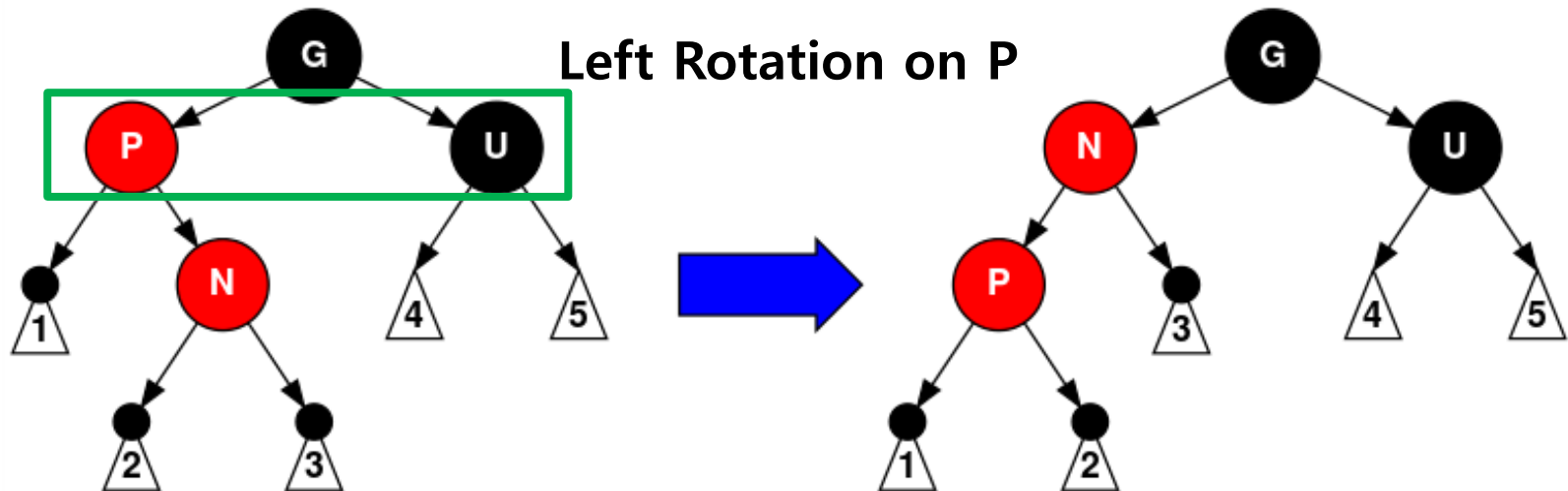
- The parent **P** is red but the uncle **U** is black; also, the current node **N** is the right child of **P**, and **P** in turn is the left child of its parent **G**.
- A *left rotation* on **P** that switches the roles of the current node **N** and its parent **P** is performed.





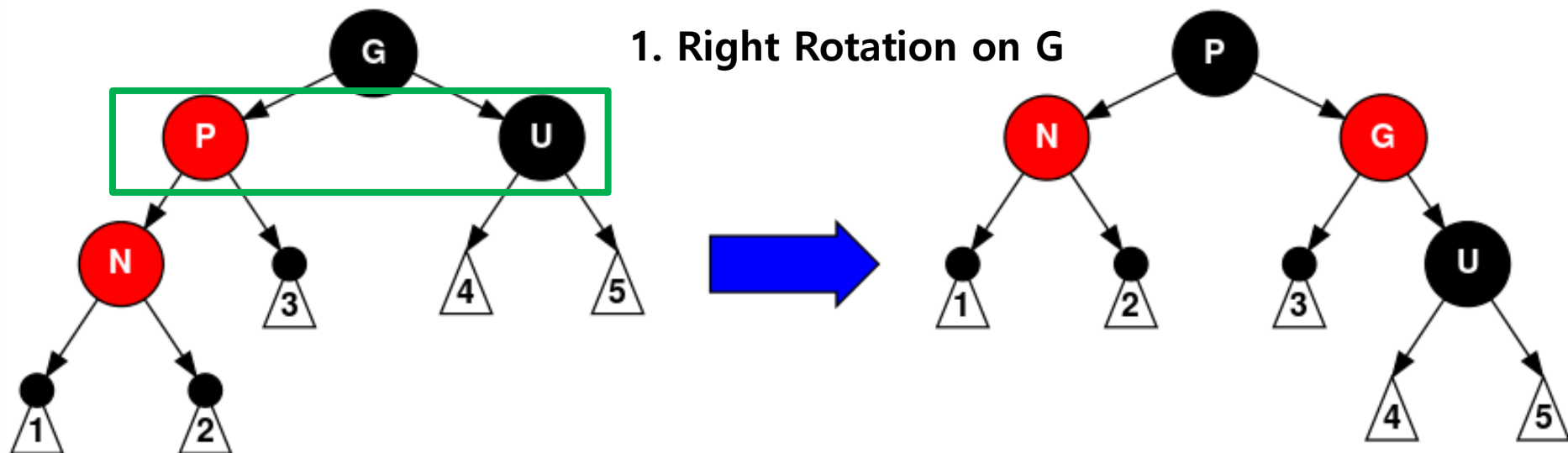
## Insertion: Case 4

- Then, the former parent node **P** is dealt with using case 5 (relabeling **N** and **P**) because Red property is still violated.
- Depth Property is not violated by the rotation.
  - The rotation causes some paths to pass through the node **N** where they did not before. It also causes some paths not to pass through the node **P** where they did before. However, both of these nodes are red, so Depth property is not violated by the rotation.



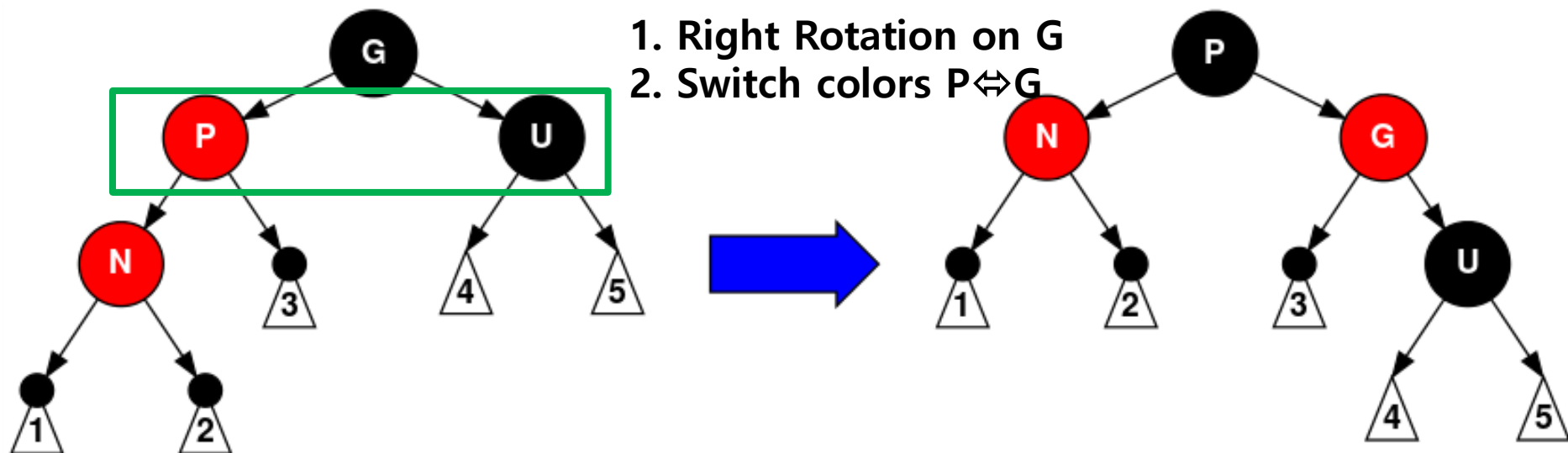
## Insertion: Case 5

- The parent **P** is red but the uncle **U** is black, the current node **N** is the left child of **P**, and **P** is the left child of its parent **G**.
- A *right rotation* on **G** is performed; the result is a tree where the former parent **P** is now the parent of both the current node **N** and the former grandparent **G**.



## Insertion: Case 5

- **G** is known to be black, since its former child **P** could not have been red otherwise (without violating Red property).
- Then, the colors of **P** and **G** are switched, and the resulting tree satisfies property.
- Depth Property also remains satisfied
  - All paths that went through any of these three nodes went through **G** before, and now they all go through **P**. In each case, this is the only black node of the three.





## Insertion: Case 4 and 5 mirrored

---

- The whole process of case 4 and 5 can be mirrored (left  $\leftrightarrow$  right)



## Exercise

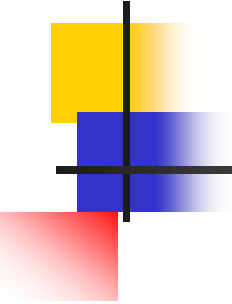
---

- Create a red-black tree by inserting the following in order:
- 13 1 25 17 11 22 6 27 15 8

# Solution: Insert 13 (case 1)

---

13



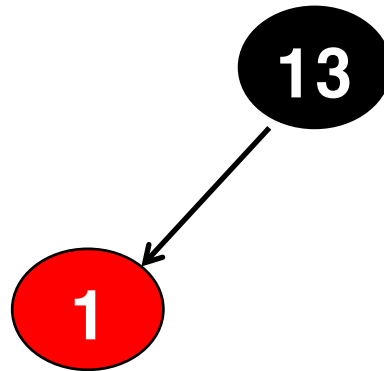
# Solution: Insert 13 (case 1)

---

13

# Solution: Insert 1 (case 2)

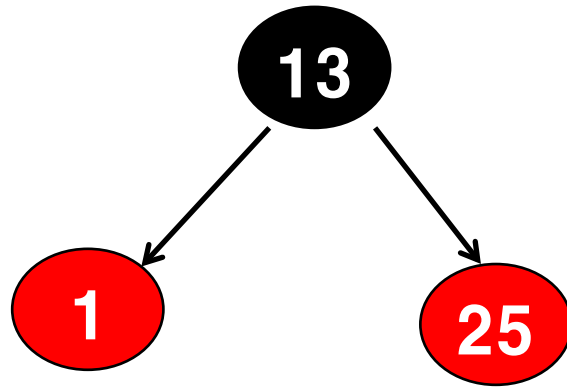
---



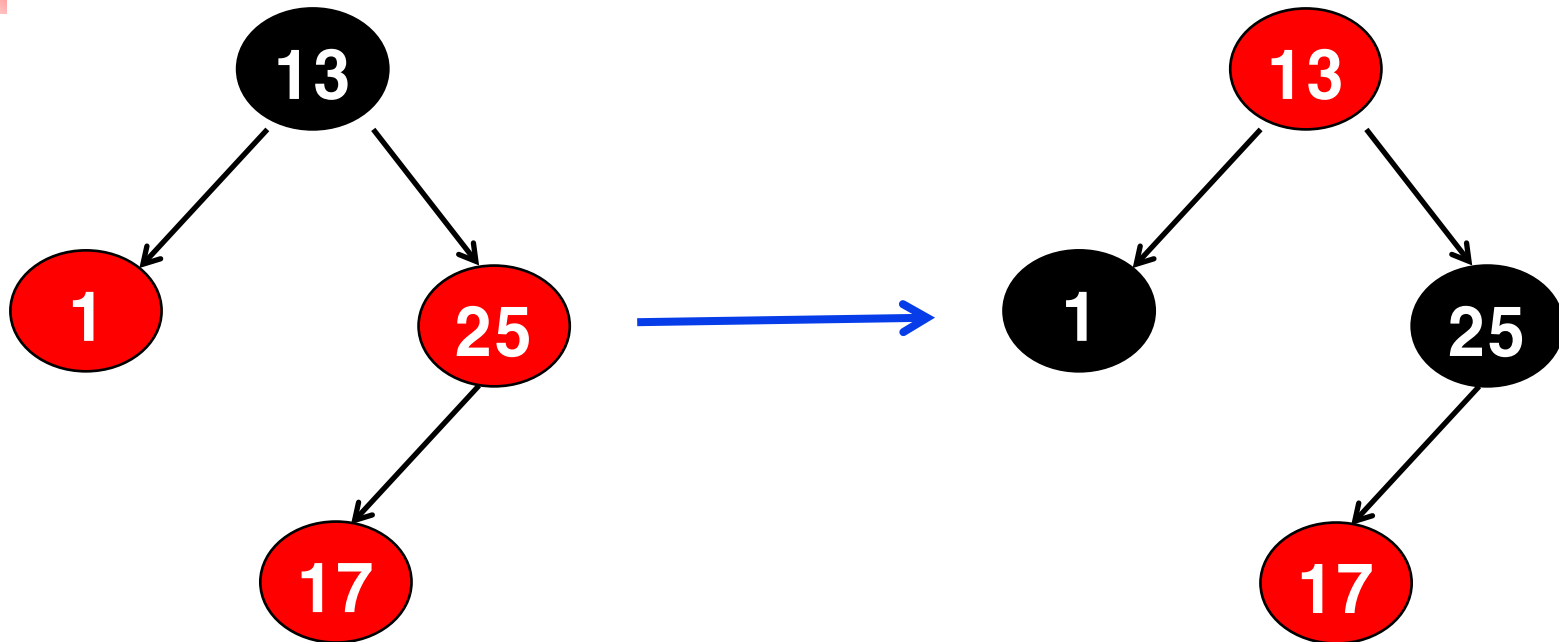


# Solution: Insert 25 (case 2)

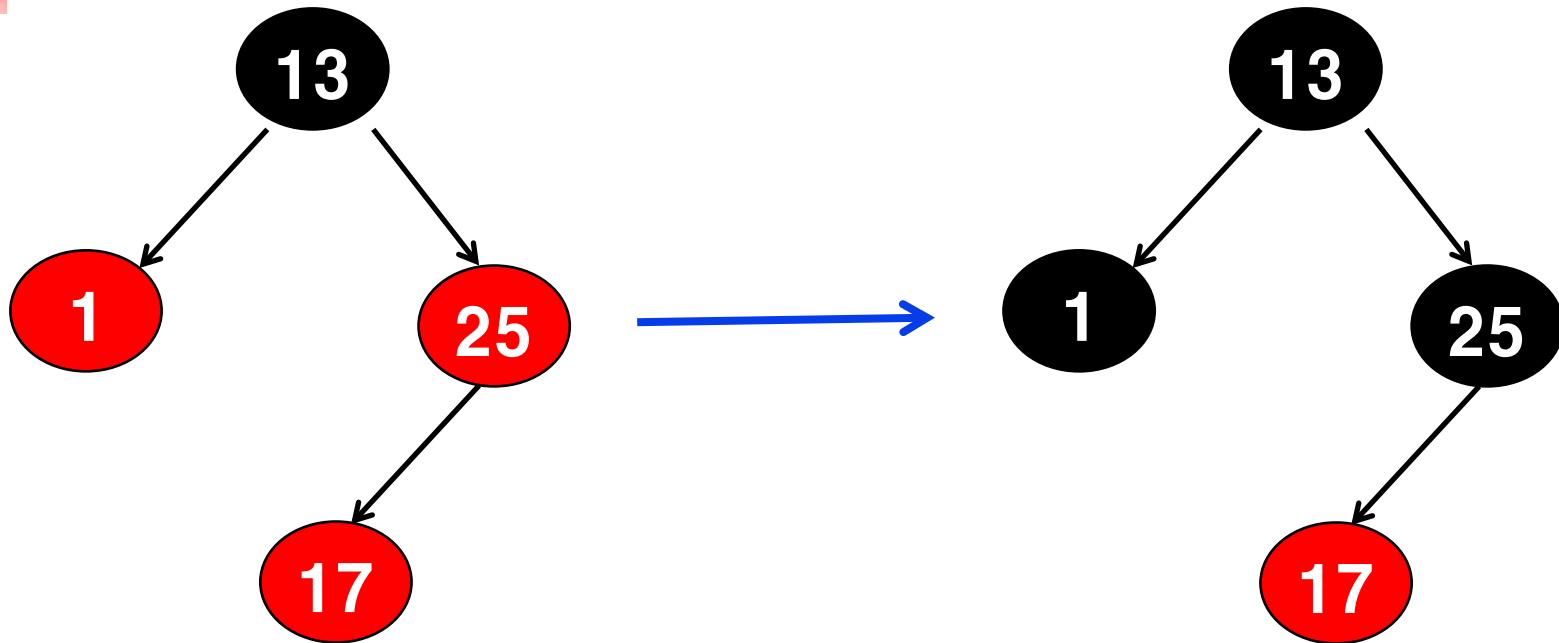
---



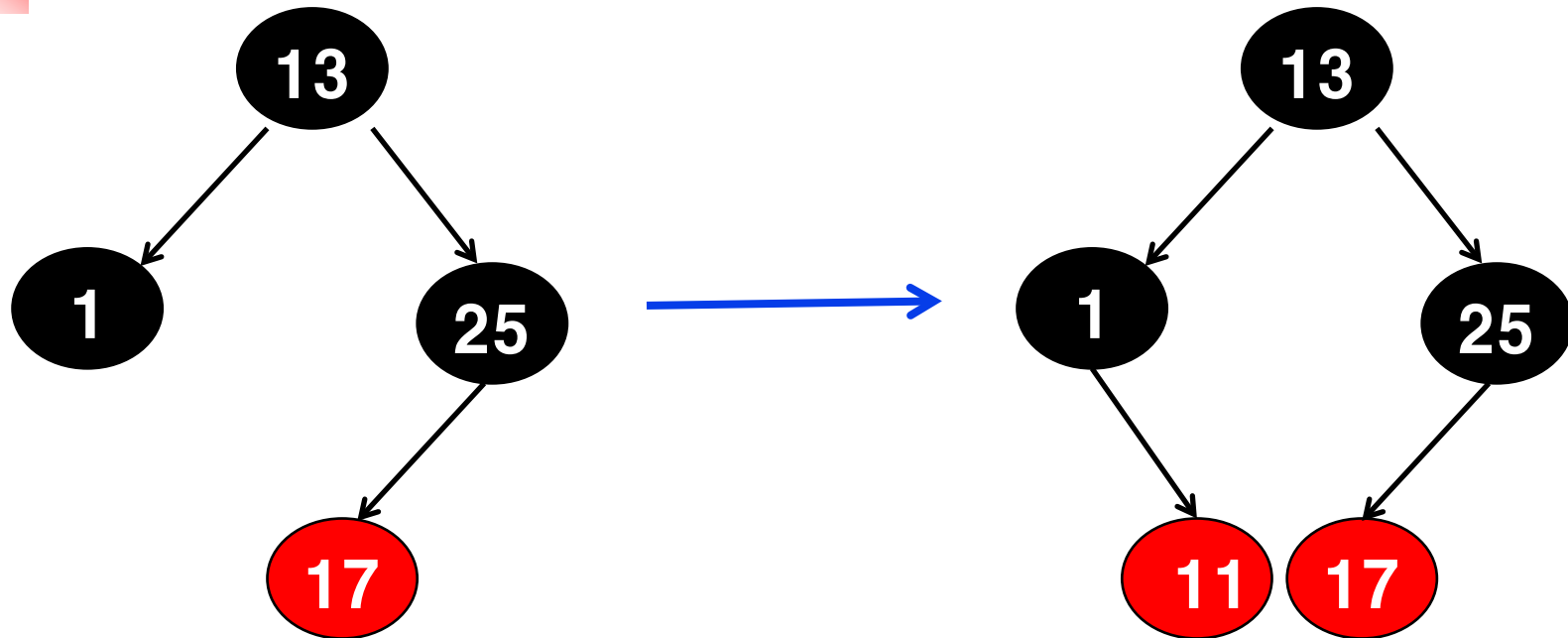
# Solution: Insert 17 (case 3)



# Solution: Insert 17 (case 1)

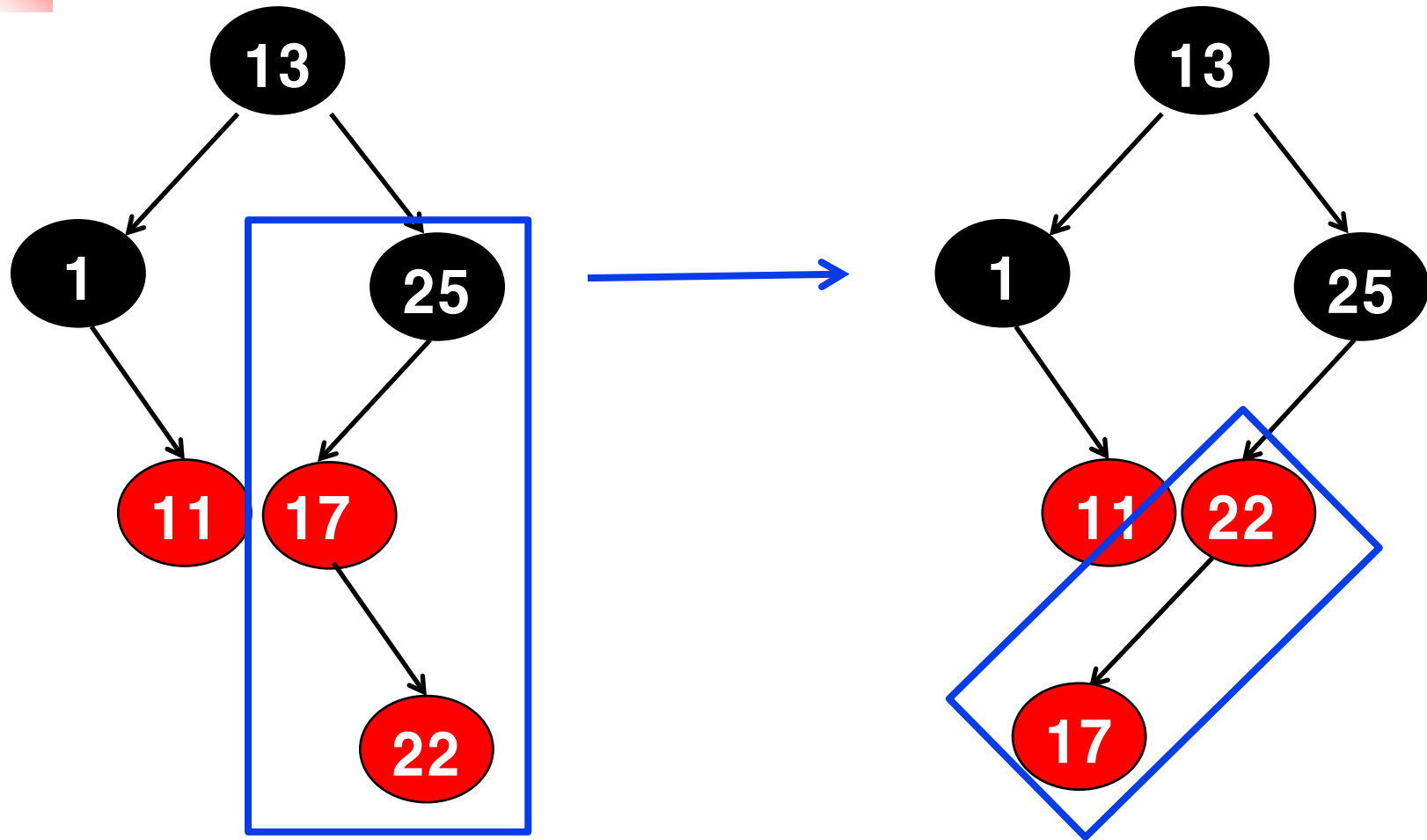


# Solution: Insert 11 (case 2)



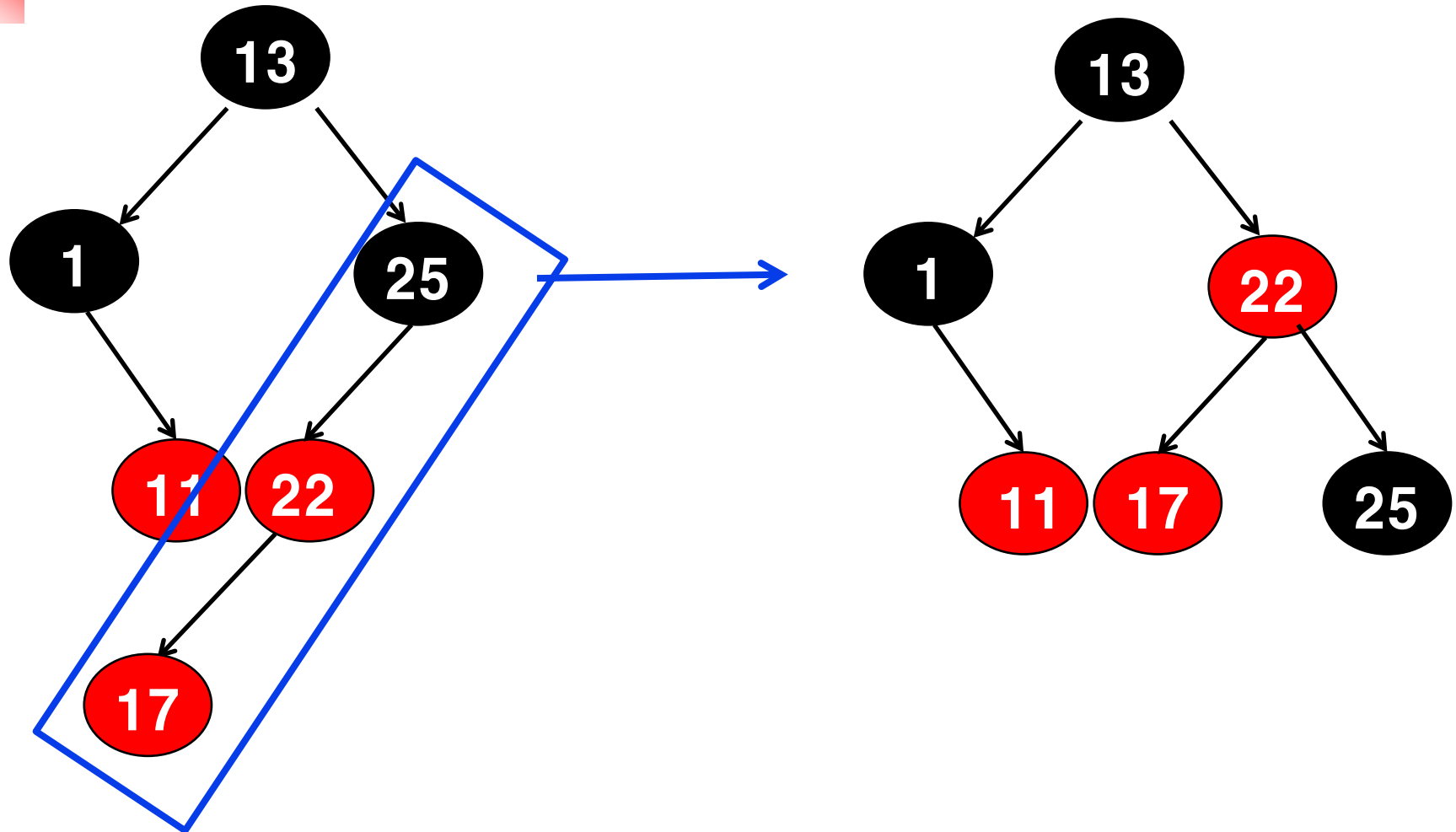
# Solution: Insert 22 (2 rotations)

## (case 4)



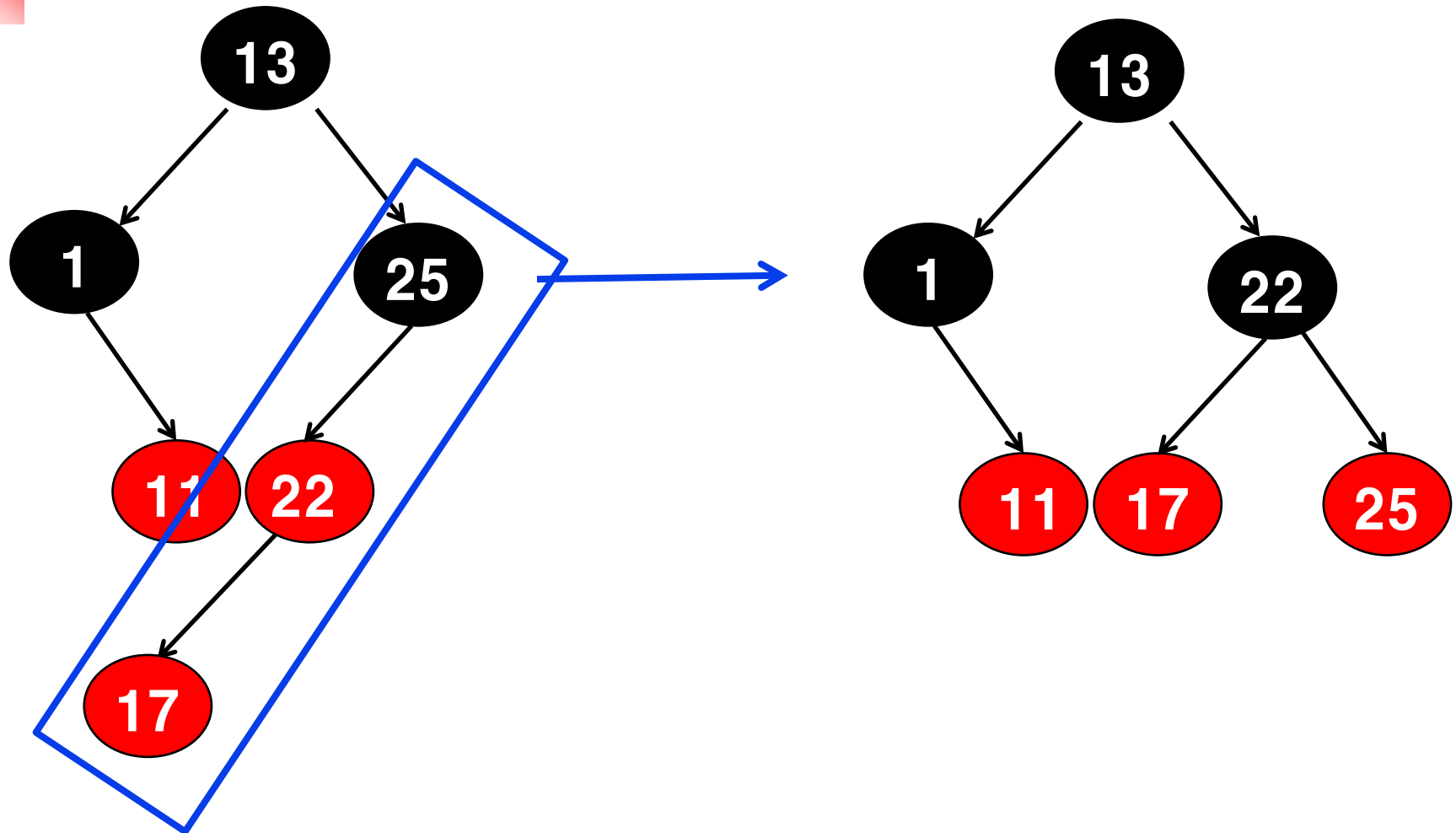
# Solution: Insert 22 (2 rotations)

## (case 5)

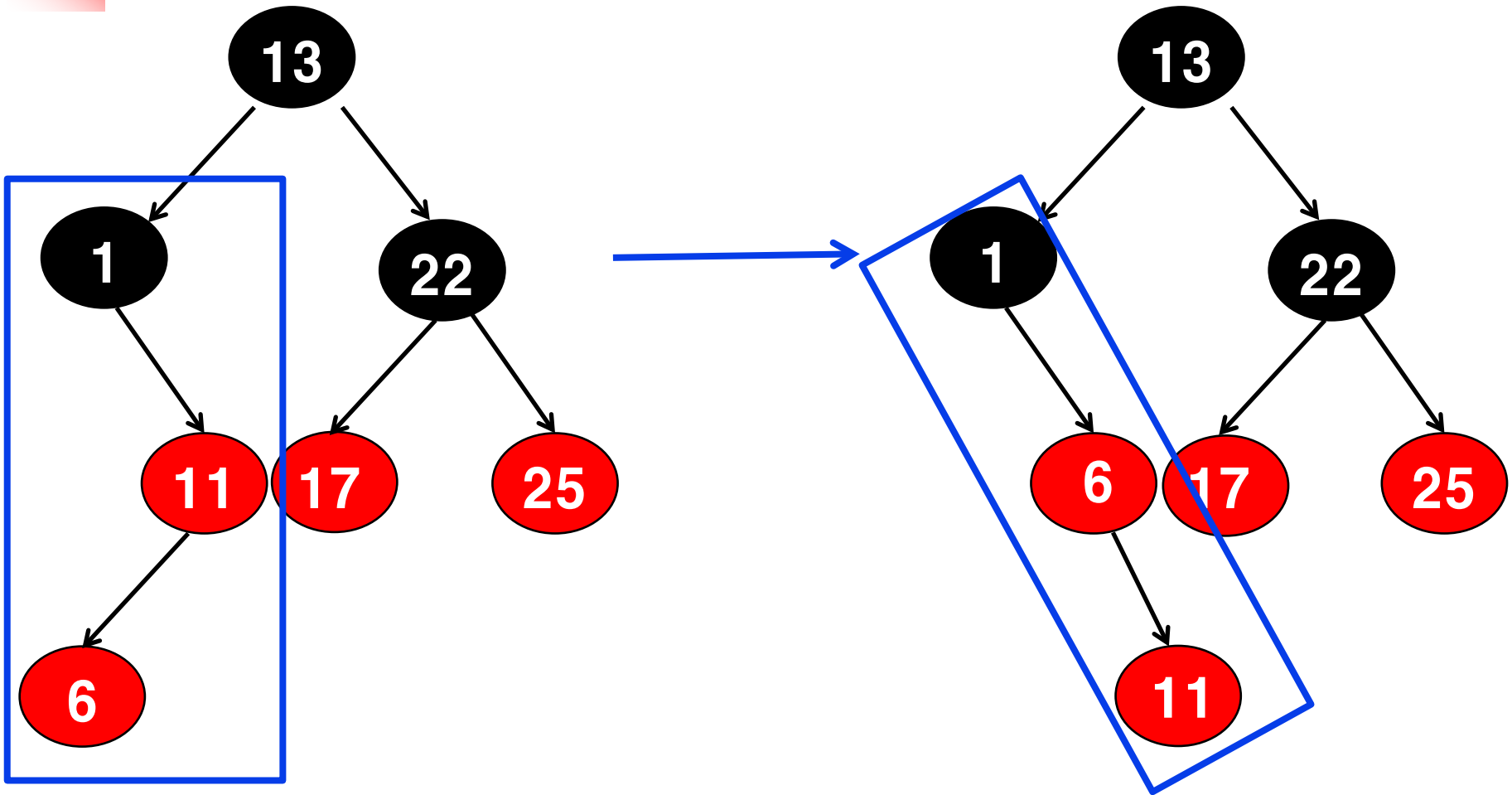


# Solution: Insert 22 (2 rotations)

## (case 5)

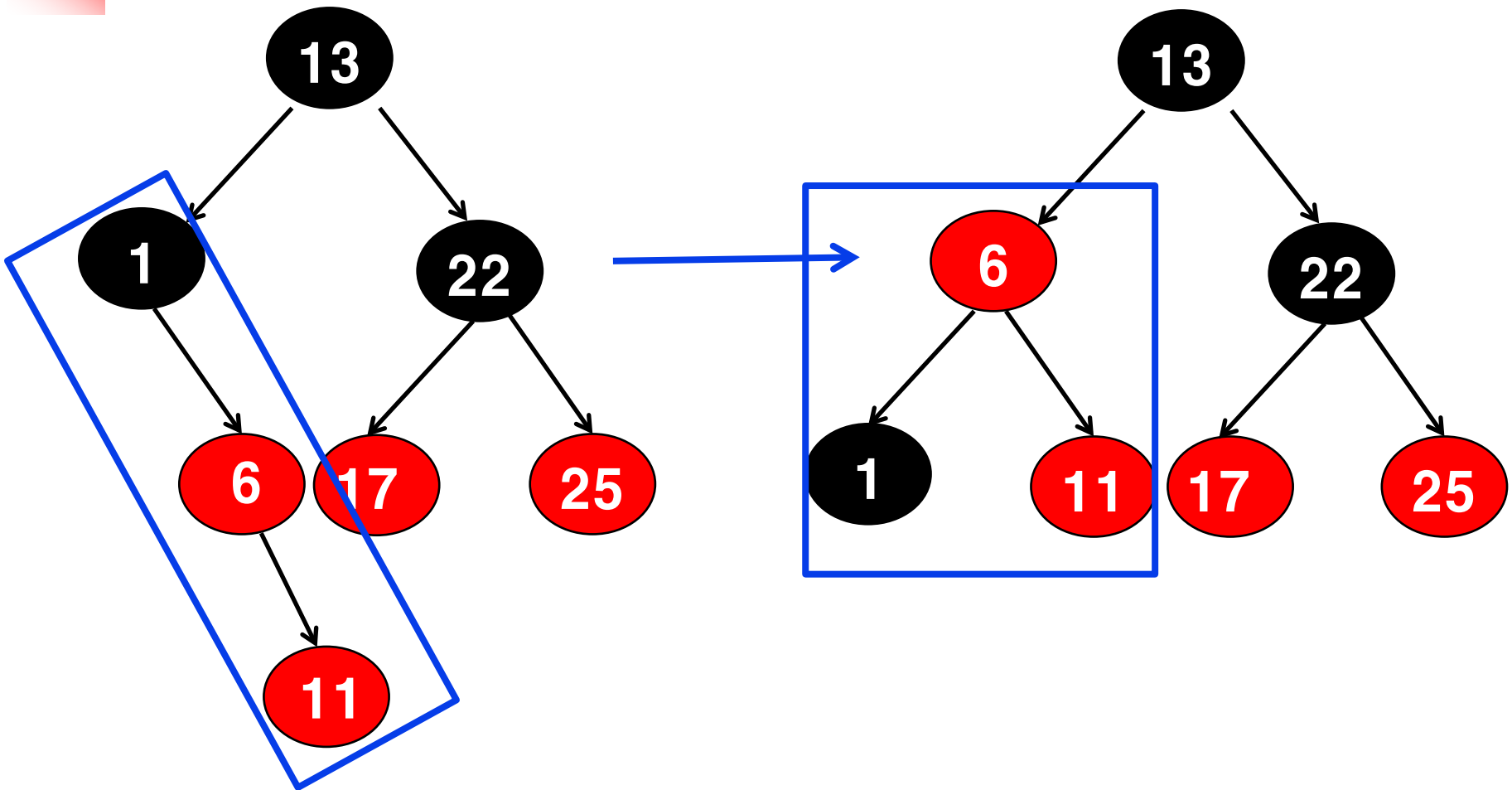


# Solution: Insert 6 (2 rotations) (case 4: mirror case)

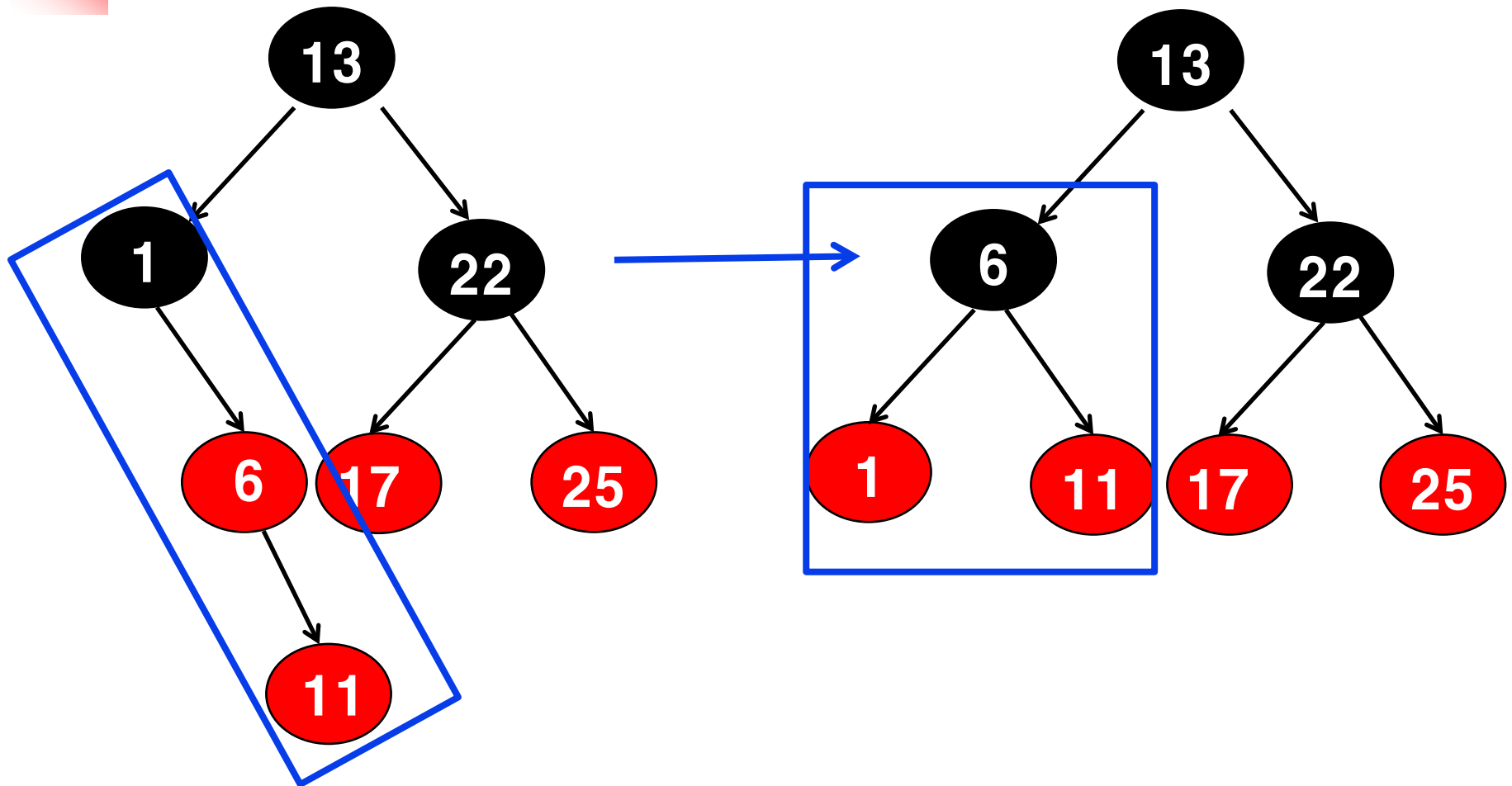




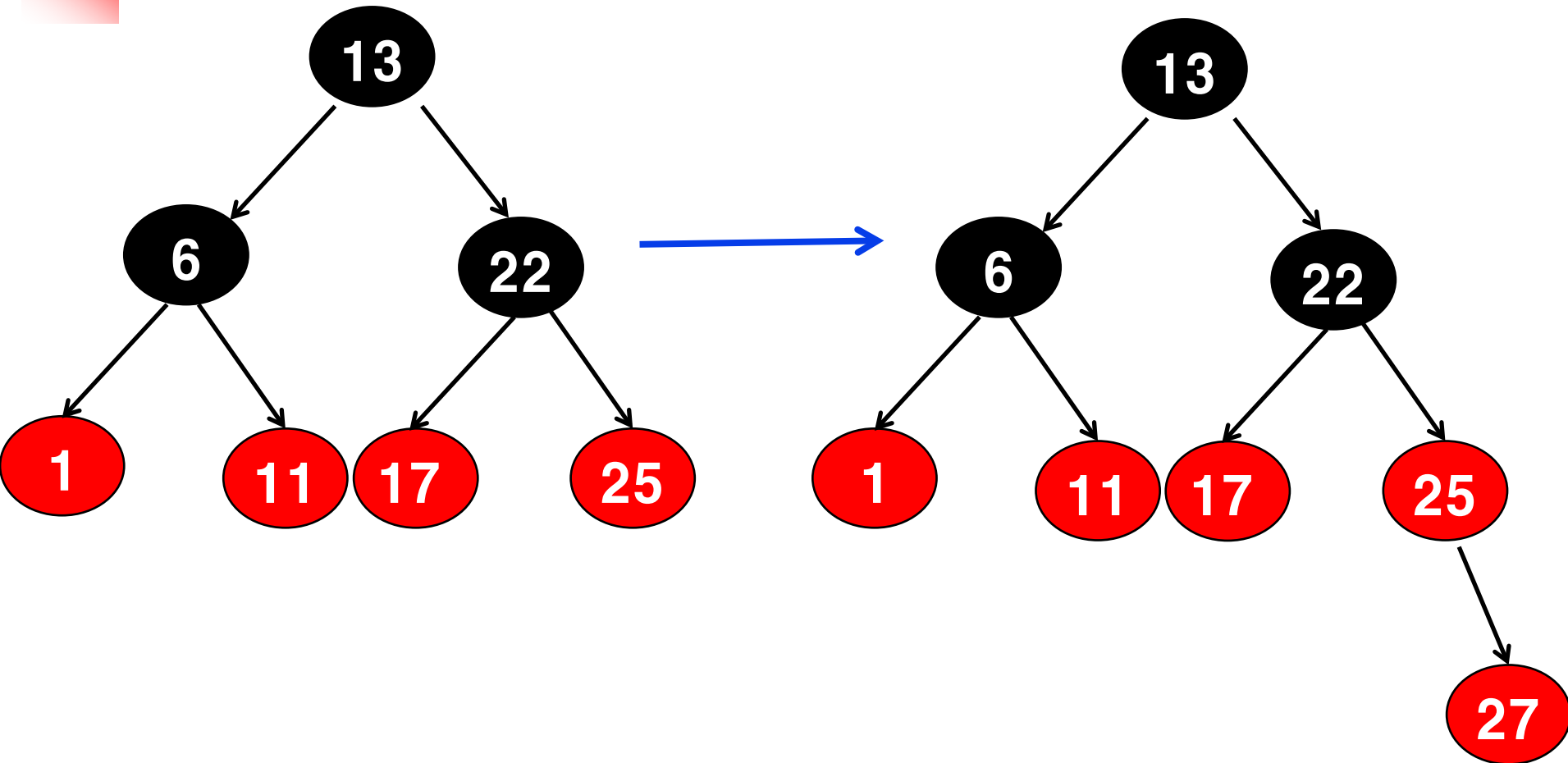
# Solution: Insert 6 (2 rotations) (case 5: mirror case)



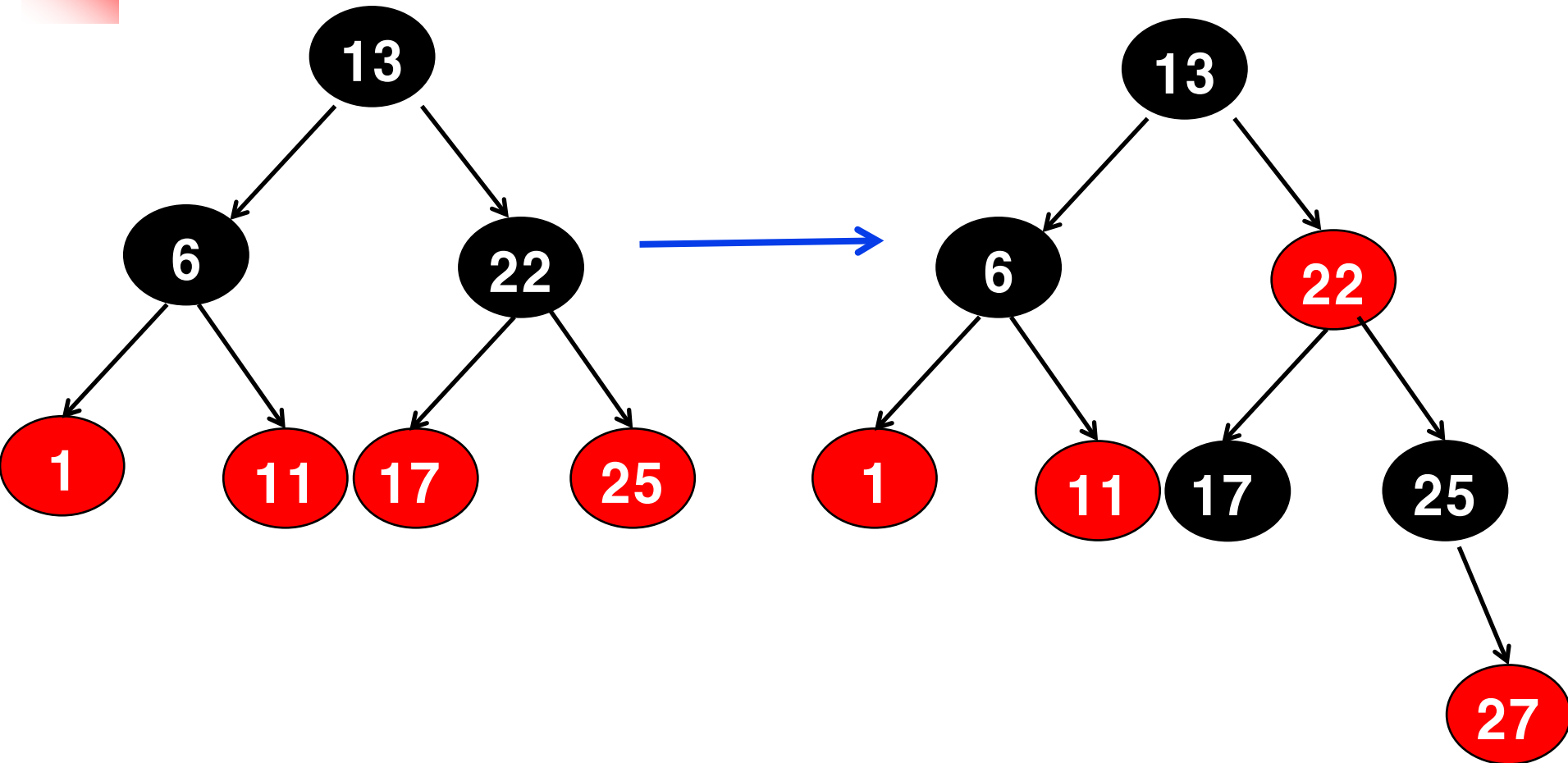
# Solution: Insert 6 (2 rotations) (case 5: mirror case)



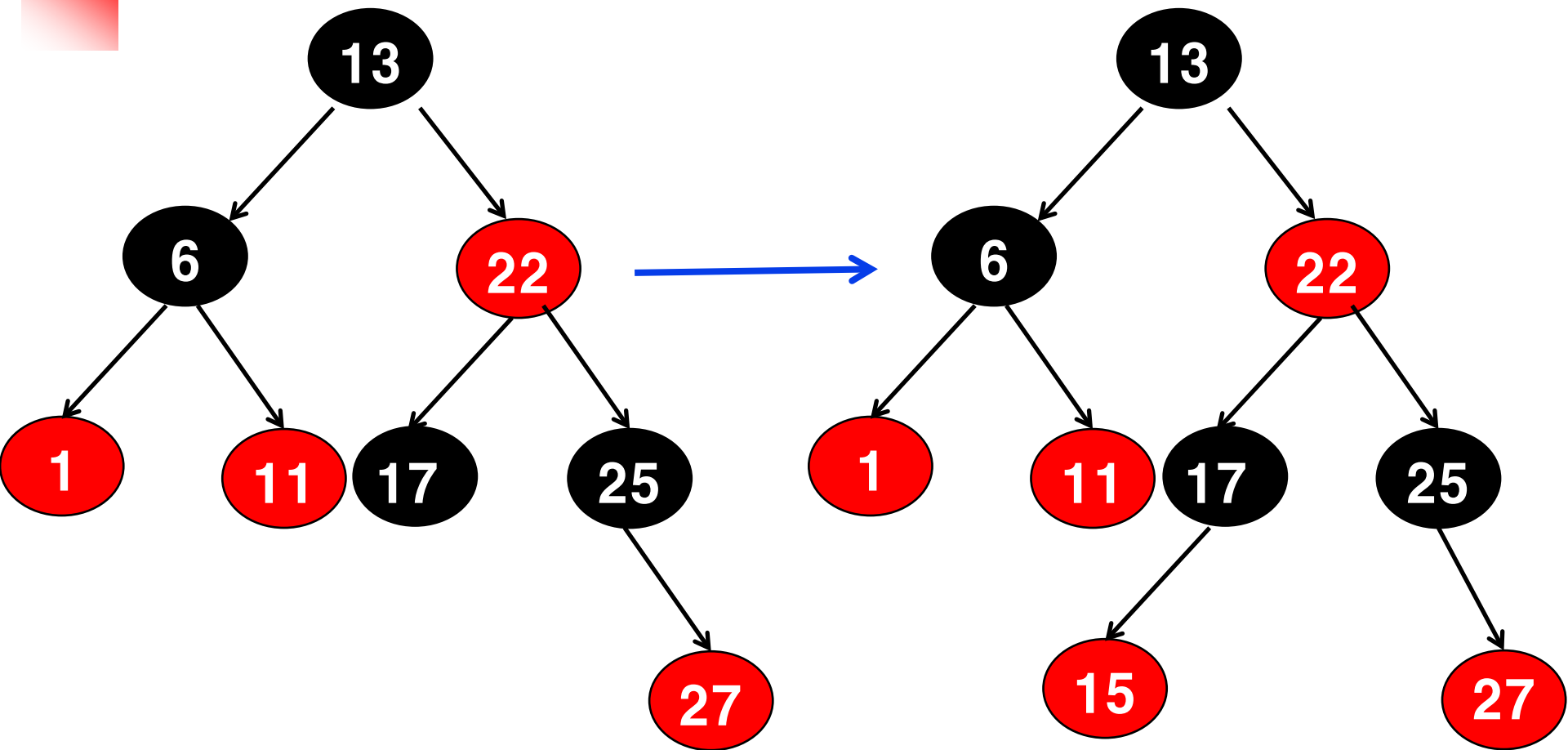
# Solution: Insert 27 (case 3)



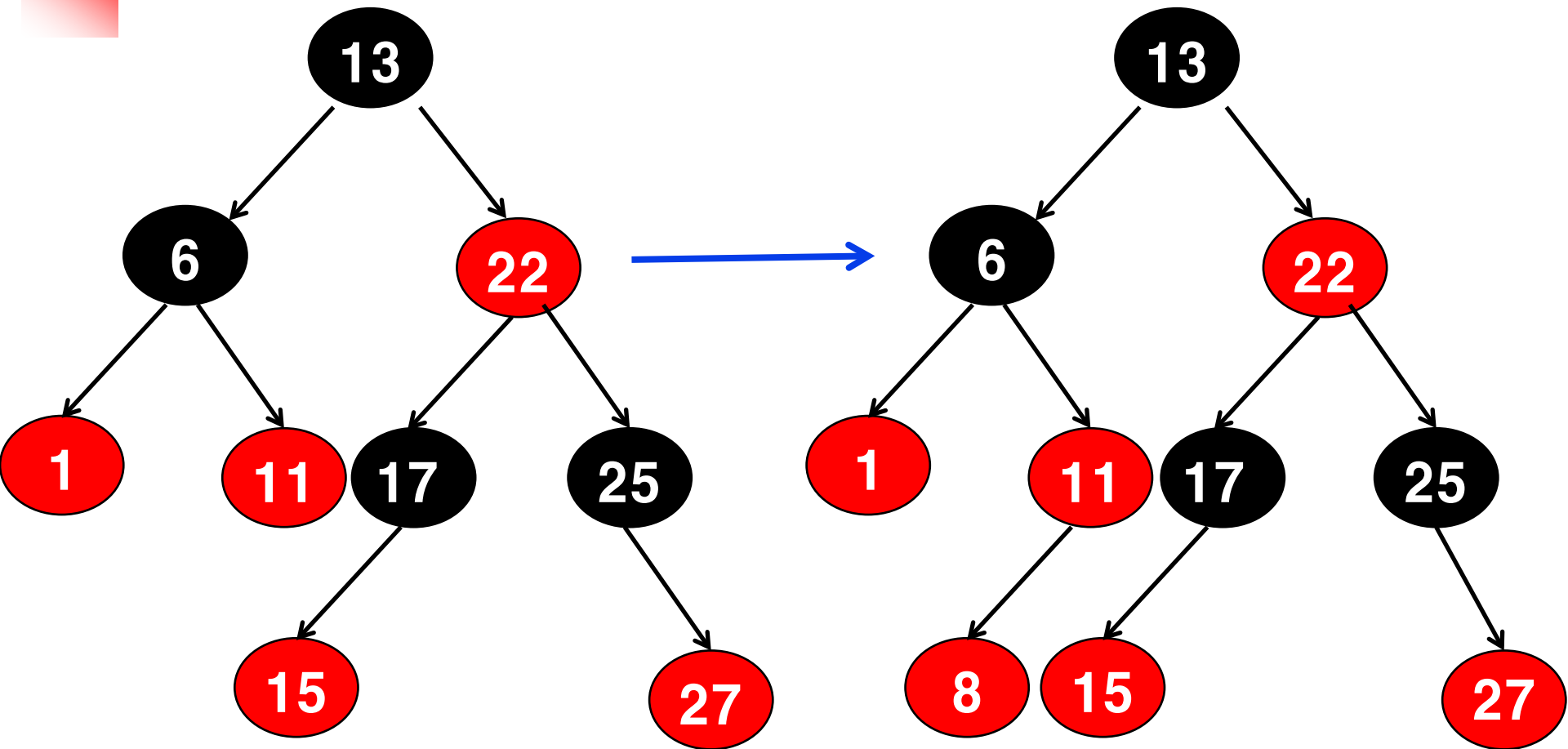
# Solution: Insert 27 (case 3)



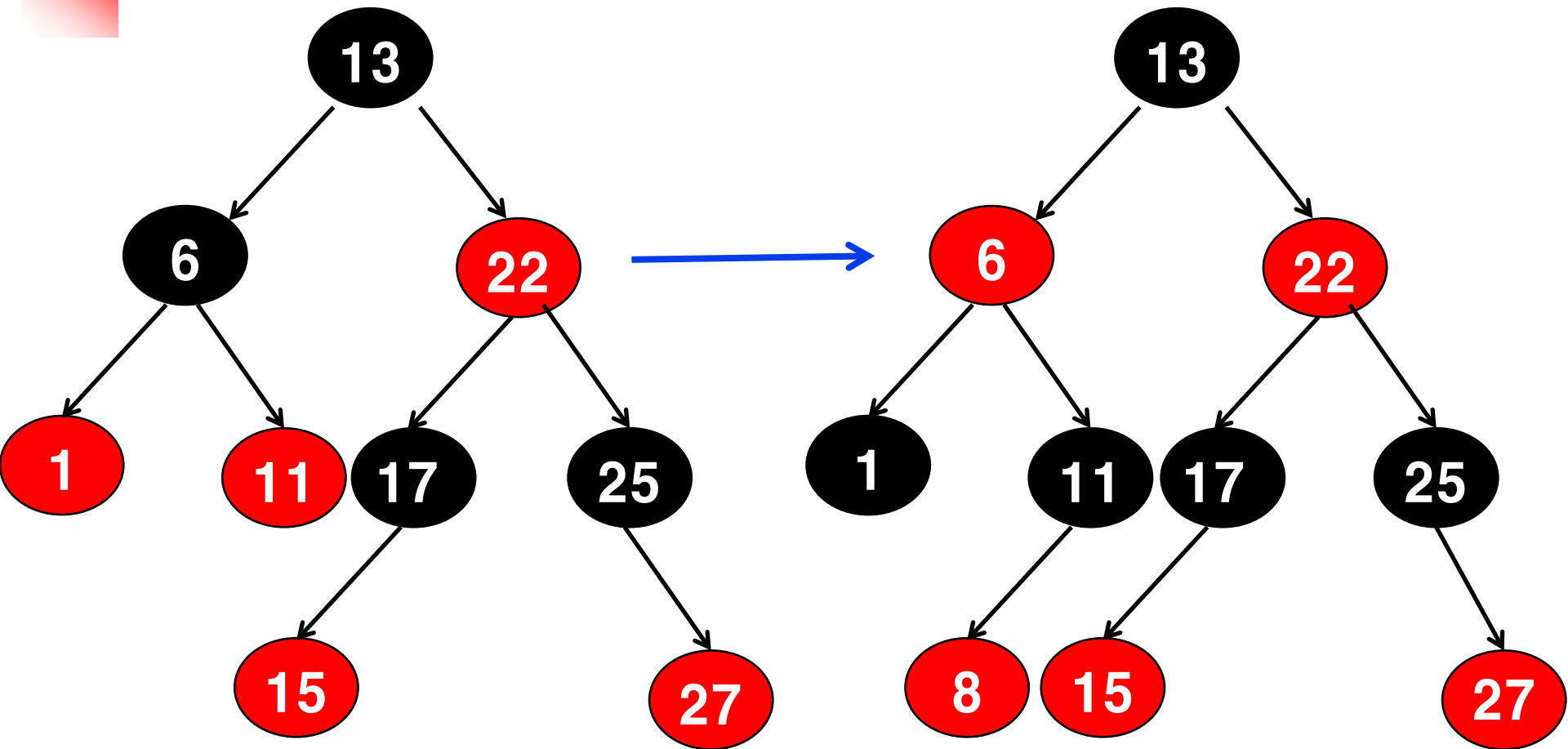
## Solution: Insert 15



# Solution: Insert 8 (case 3)



# Solution: Insert 8 (case 3)





## Deletion

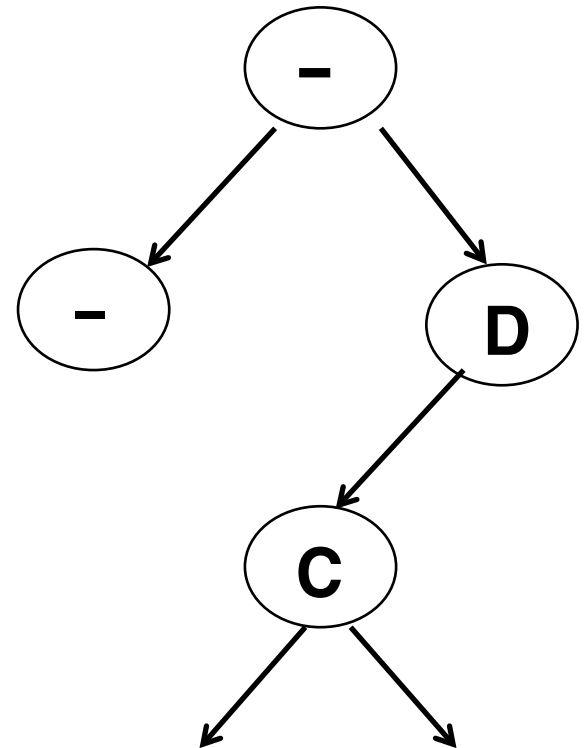
---

- Delete a node as with a regular binary search tree.
- If the deleted node was red, we are done.
- If a black node is deleted and replaced by a black child, the child is marked as a **double black (or extra black)** node.
- To rebalance the tree, the main task is to convert the extra black node to a single (normal) black node.



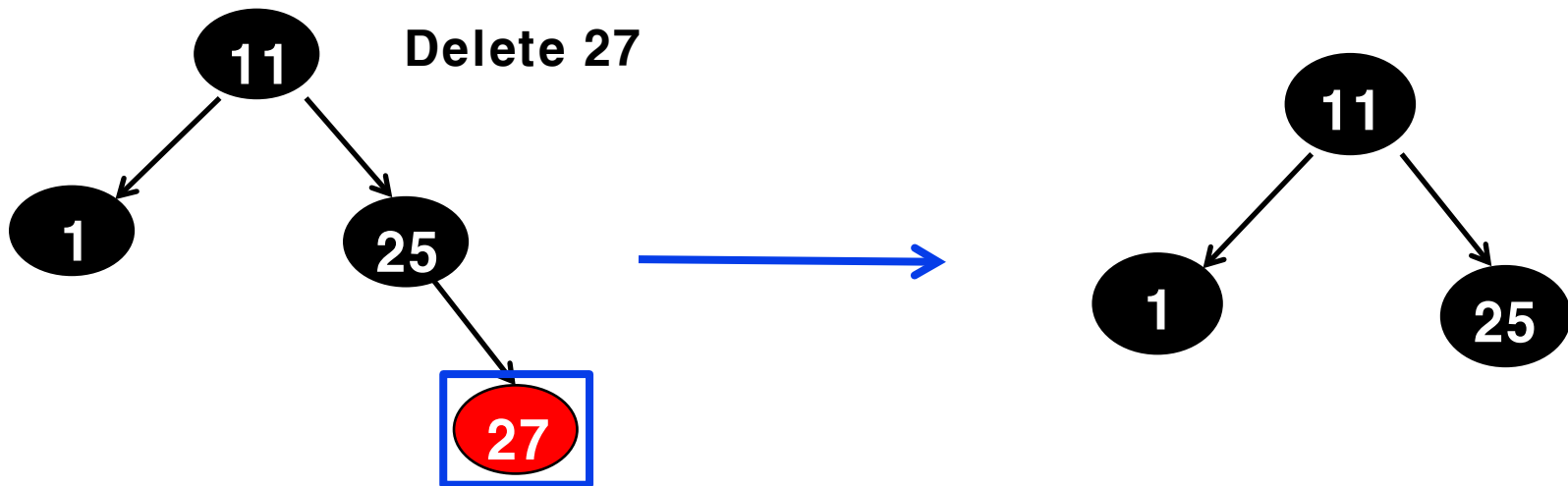
# Notation

- **D**: a node to be deleted by BST deletion
- **C**: a selected child of **D**



## Case 1

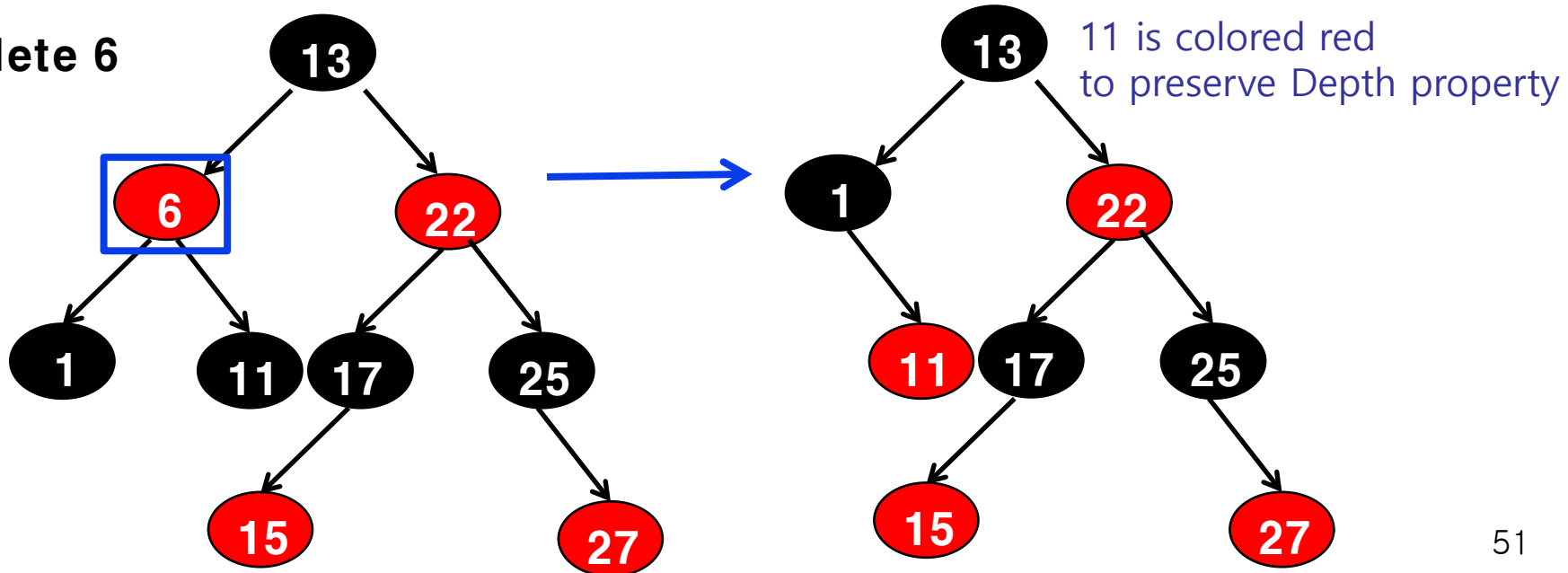
- If **D** is a red & a leaf.
  - Simply delete **D**.



## Case 1

- If **D** is a **red** & a non-leaf.
  - Simply replace it with its child **C**

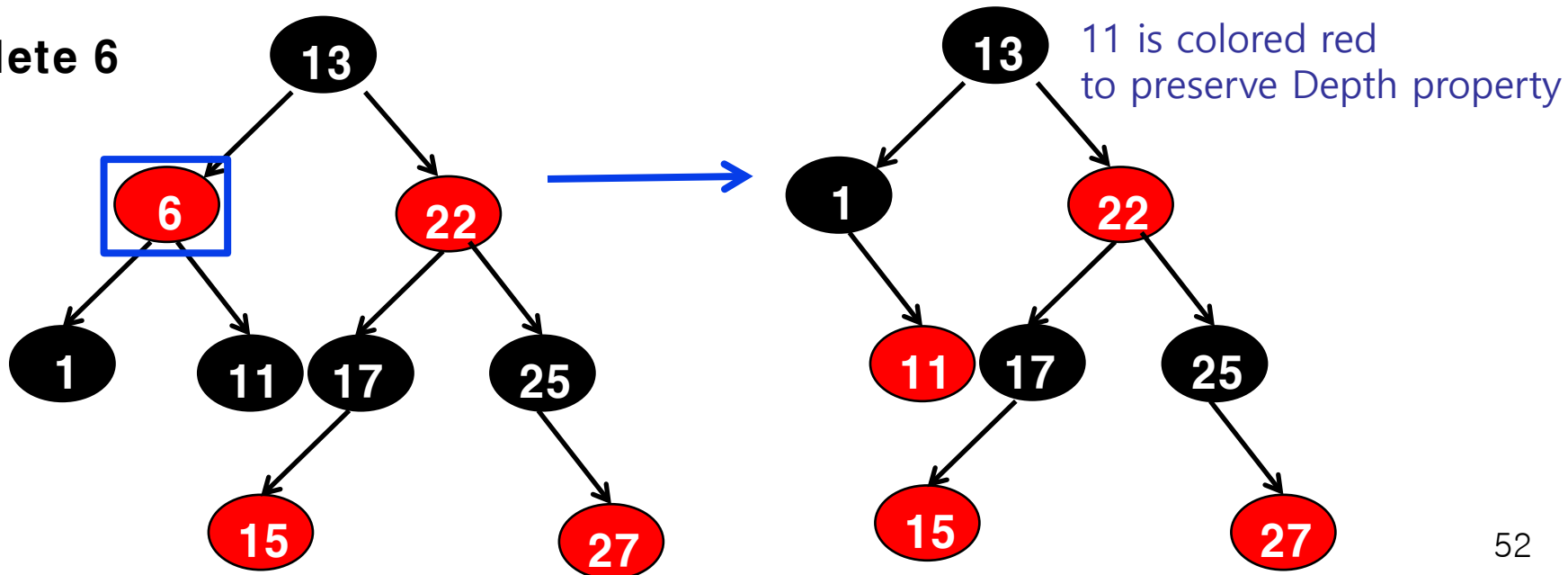
Delete 6



# Case 1

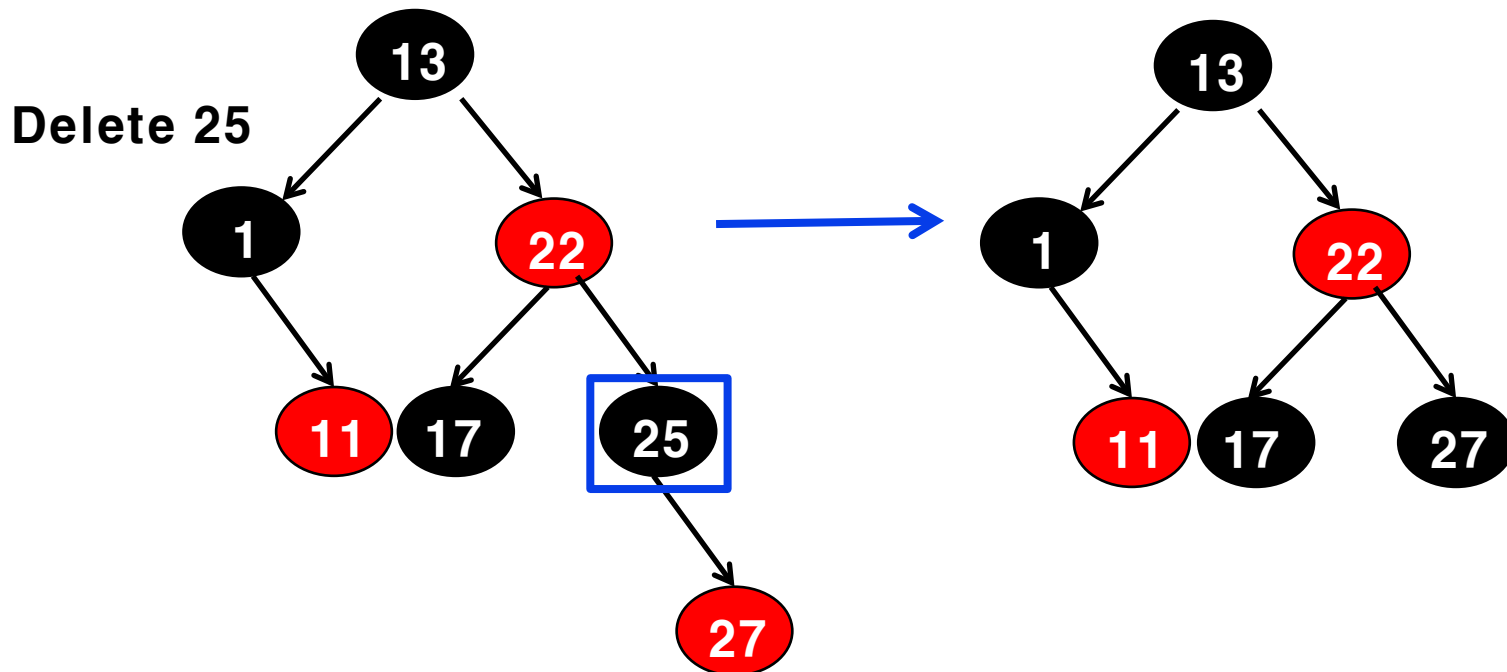
- **C** must be black by Red property.
  - (This can only occur when **D** has two leaf children, because if the red node **D** had a black non-leaf child on one side but just a leaf child on the other side, then the count of black nodes on both sides would be different, thus the tree would violate Depth property.)
- Leaf Property and property Red still hold.
  - All paths through the deleted node will simply pass through one fewer red node, and both the deleted node's parent and child must be black.

**Delete 6**



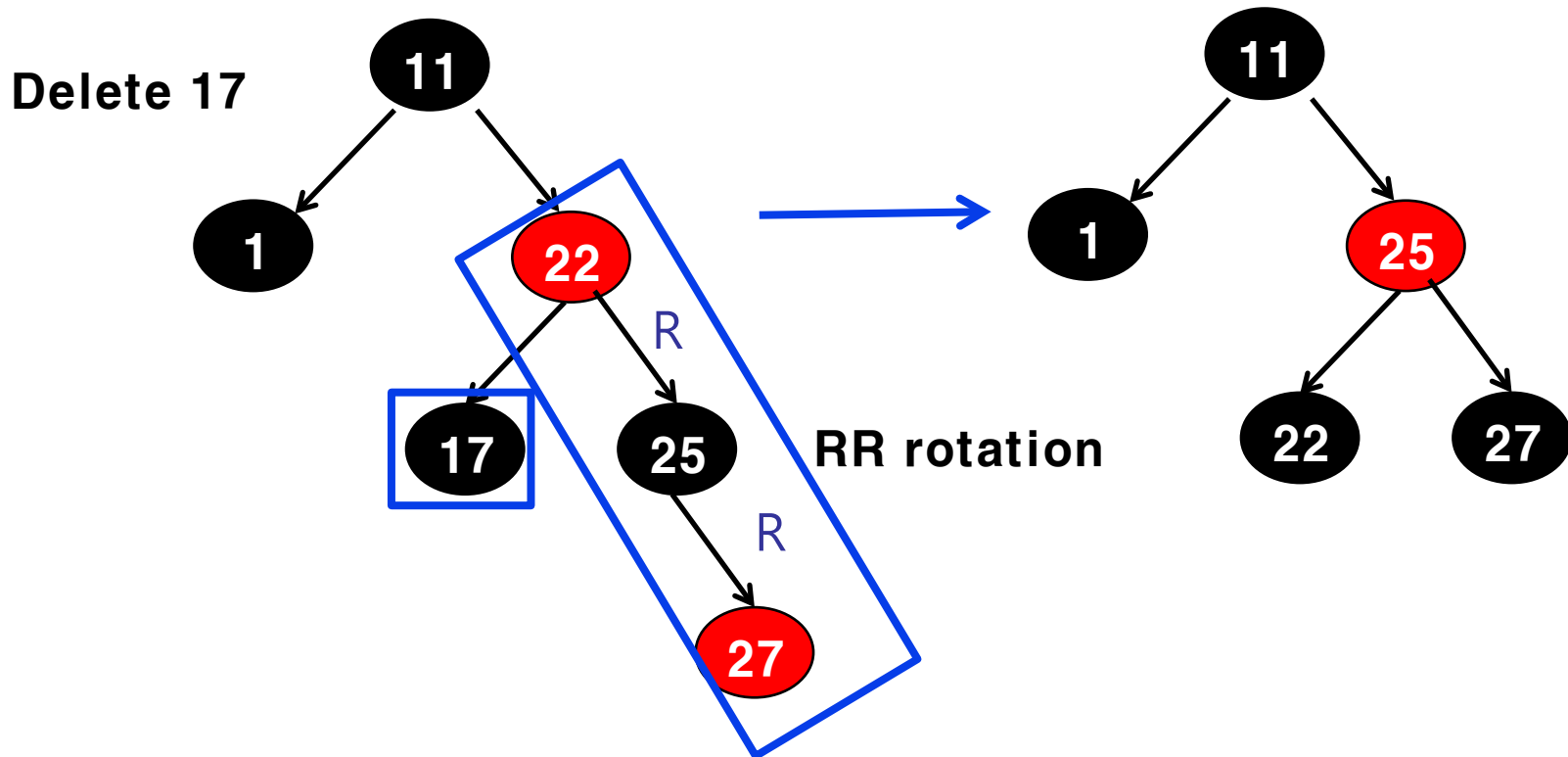
## Case 2

- **D** is black and **C** is red.
  - Delete **D**.
  - Replace it with **C**.
  - Recolor **C** black.



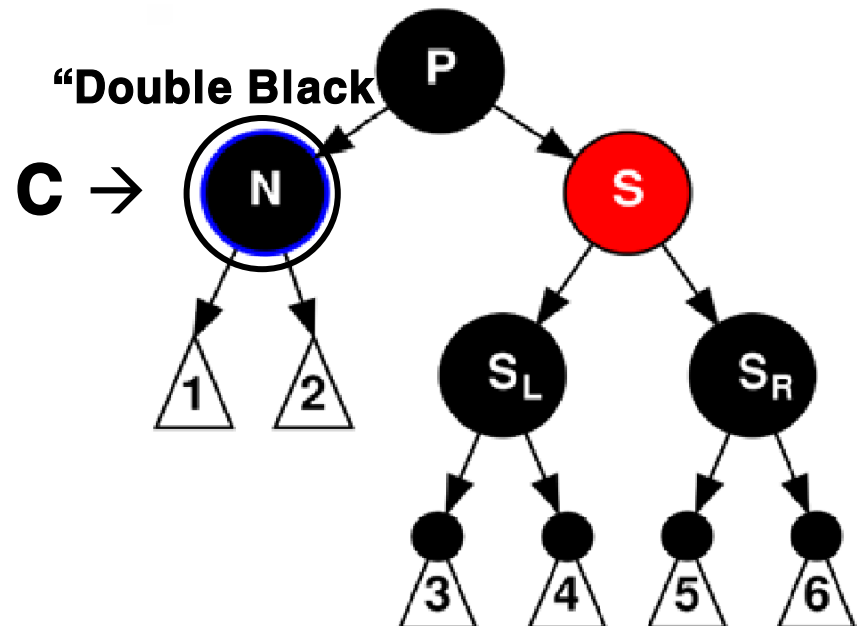
## Medium Case: parent P is red, D' s sibling is black, and D' s sibling' s child is red

- Left rotate at D's sibling, and exchange colors of P, D's sibling and D's sibling's child.
- \*\* Note the similarity to AVL Tree rotation



## Case 3: Complex Cases

- Both **D** and **C** are black.
- Delete **D**, and replace it with **C**.
- Relabel **C** (in its new position) as **N**, and its sibling (its new parent's other child) **S**. (**S** was previously the sibling of **D**.)
- **N** is double black.





## Case 3 (contd.)

---

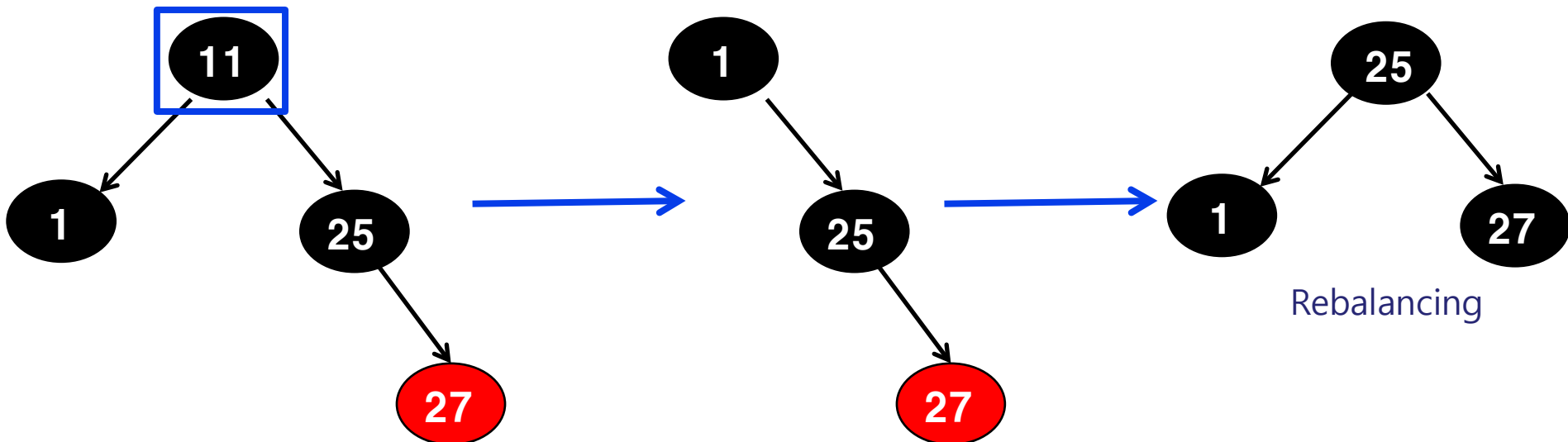
- Double black needs to be removed to single black to restore balance.
  - This causes paths have one fewer black node than paths that do not.
- As this violates Depth property, the tree must be rebalanced.
  - There are several cases to consider:



## Case 3-1

- **N** is a new root.
- Remove double black to single black.
- Terminal state (Done!)
  - One black node has been deleted from every path, and the new root is black, so the properties are preserved.

Delete 11

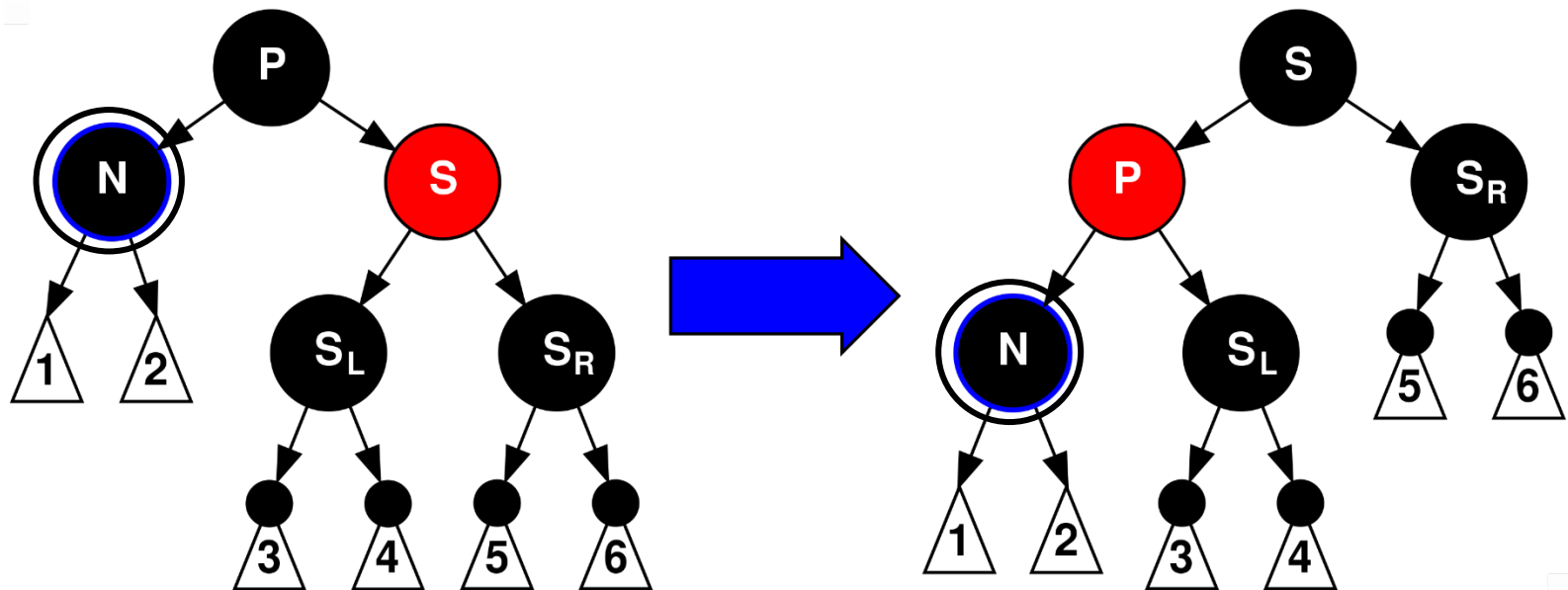


Depth Property is violated.

Rebalancing

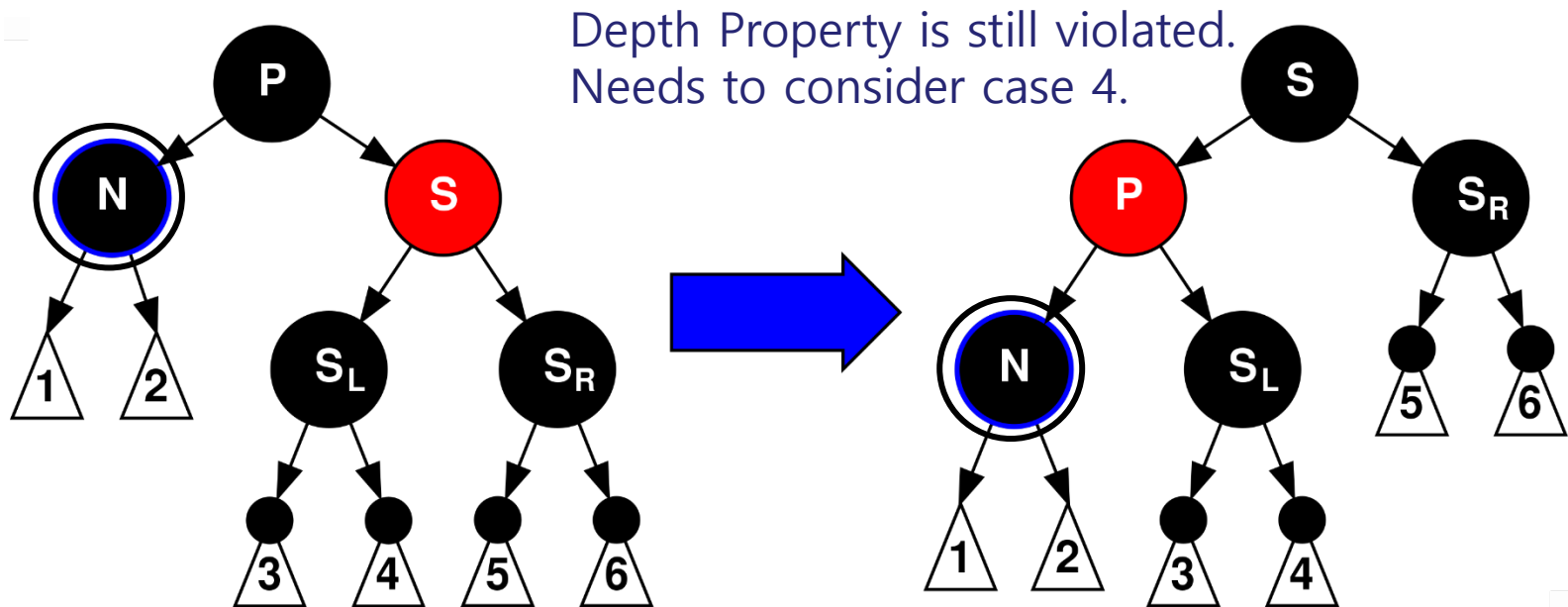
## Case 3-2

- Sibling **S** is red.
- Parent **P** and both of **S**'s children **S<sub>L</sub>** and **S<sub>R</sub>** must be black to be a RB-tree.
- Switch the colors of **P** and **S**, and then *rotate toward N* at **P**.



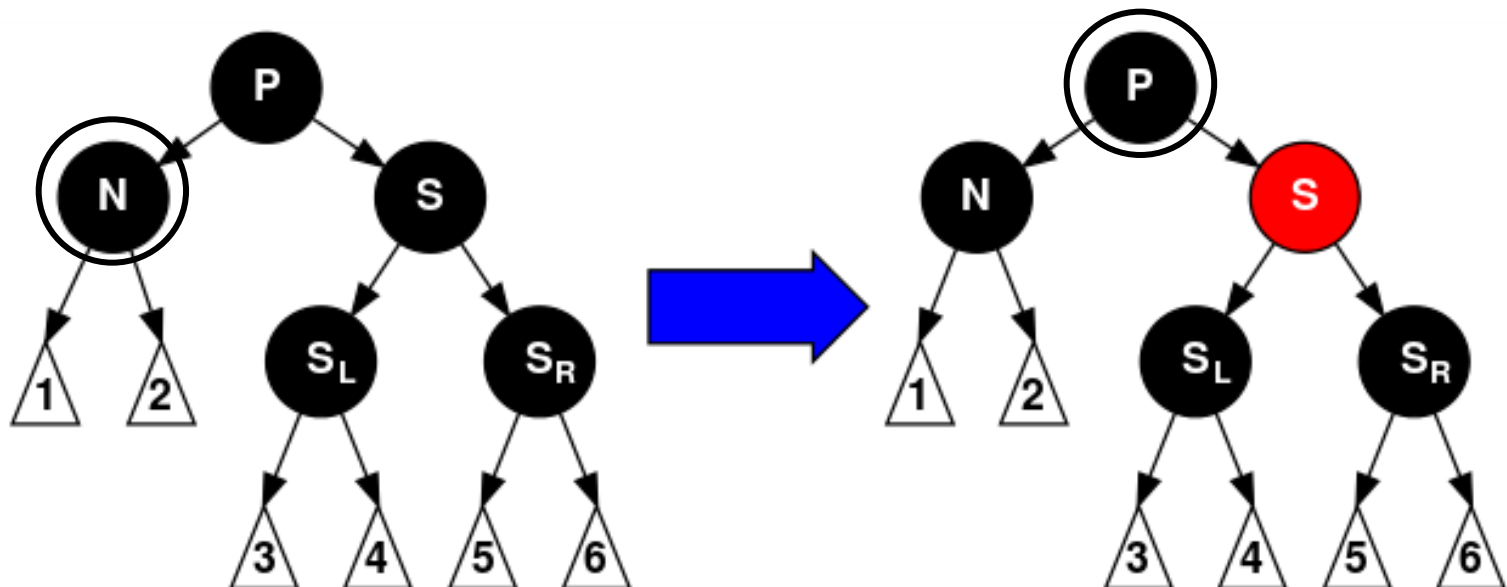
## Case 3-2 (contd.)

- **N** is still double black.
- Now **N** has a black sibling and a red parent, so we can proceed to cases 4, 5, or 6.



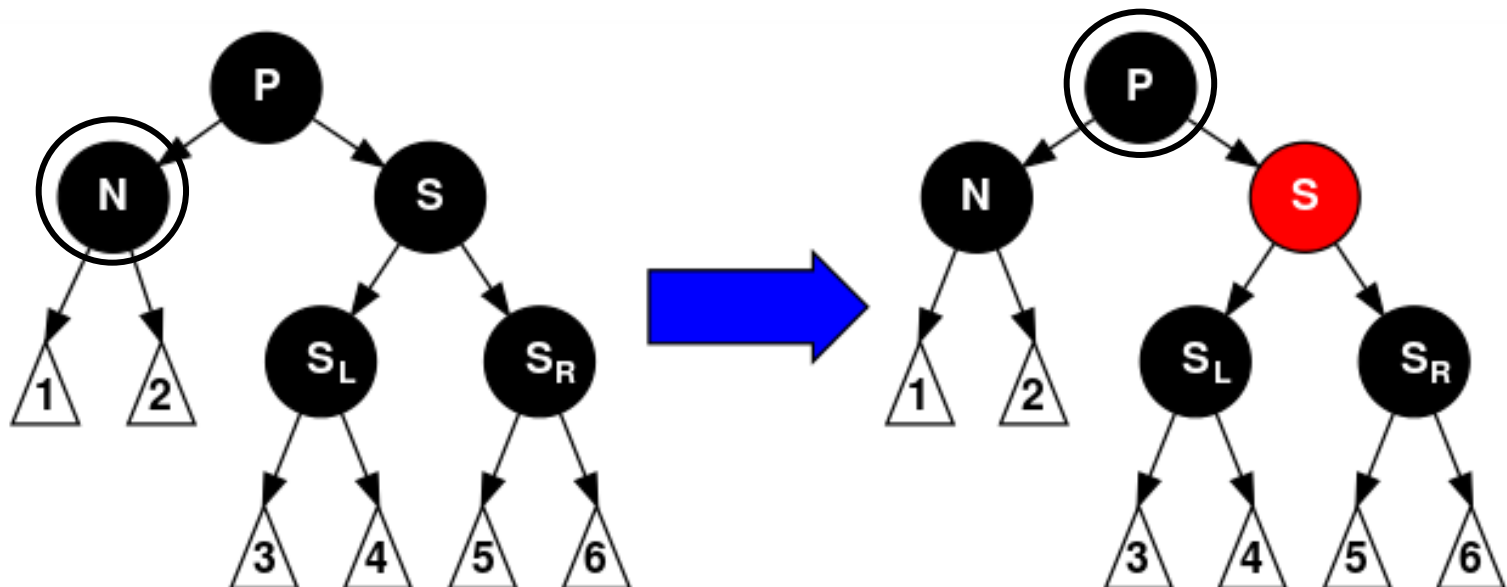
## Case 3-3

- Parent **P**, Sibling **S**, **S**'s children **S<sub>L</sub>** and **S<sub>R</sub>** are all black.
- Recolor **S** red.
- Move double black to **P**.



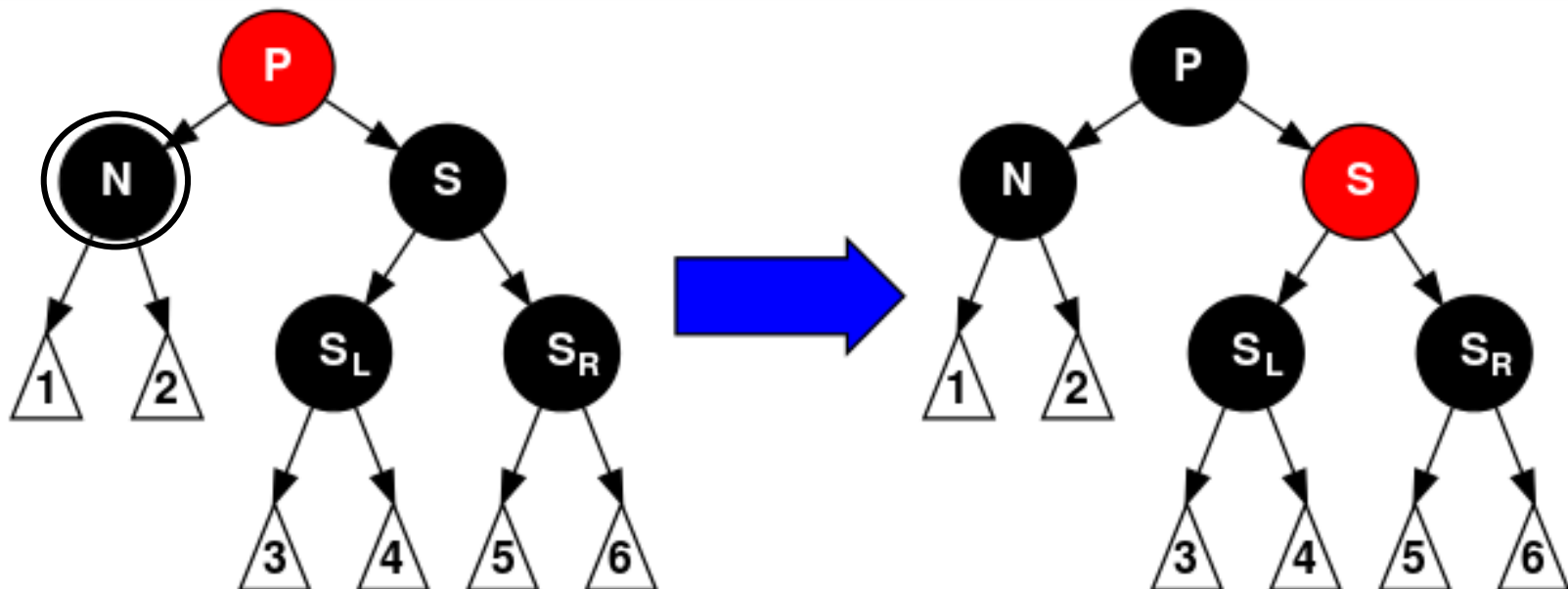
## Case 3-3 (contd.)

- The result is that all paths passing through **S** have one less black node.
- Because deleting **N**'s original parent made all paths passing through **N** have one less black node, this evens things up.
- However, all paths through **P** now have one fewer black node than paths that do not pass through **P**, so Depth property is still violated. To correct this, we perform the rebalancing procedure on **P**, starting at case 1.



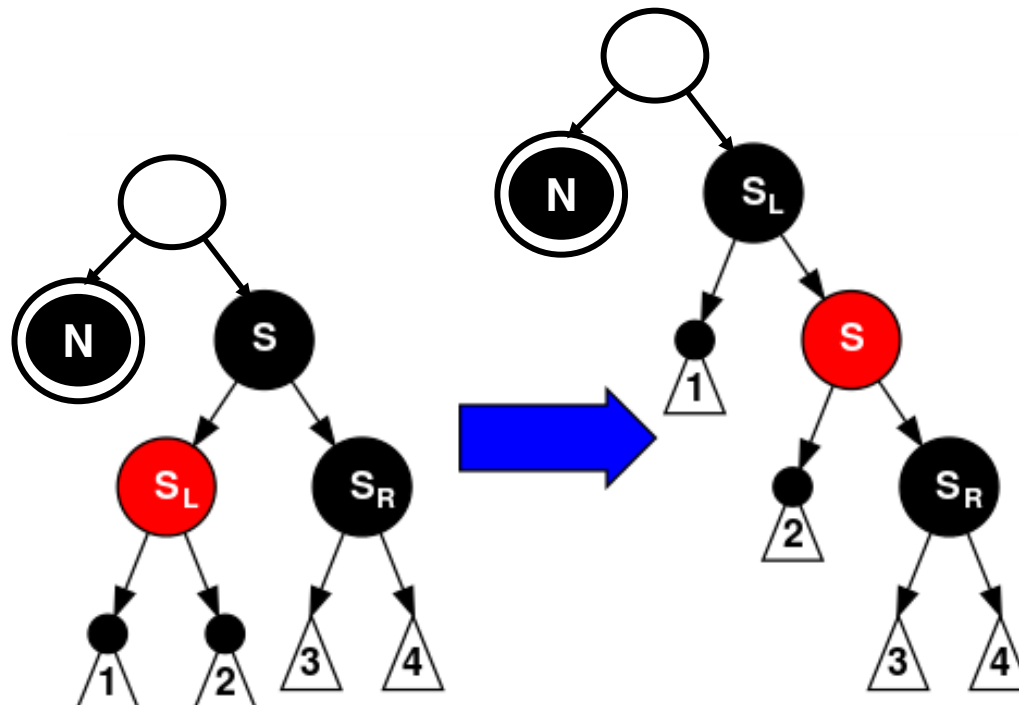
## Case 3-4

- Parent **P** is red, but Sibling **S**, **S**'s children **S<sub>L</sub>** and **S<sub>R</sub>** are black.
- Simply exchange the colors of **S** and **P**.
- Terminal state.
  - This does not affect the number of black nodes on paths going through **S**, but it does add one to the number of black nodes on paths going through **N**, making up for the deleted black node on those paths.



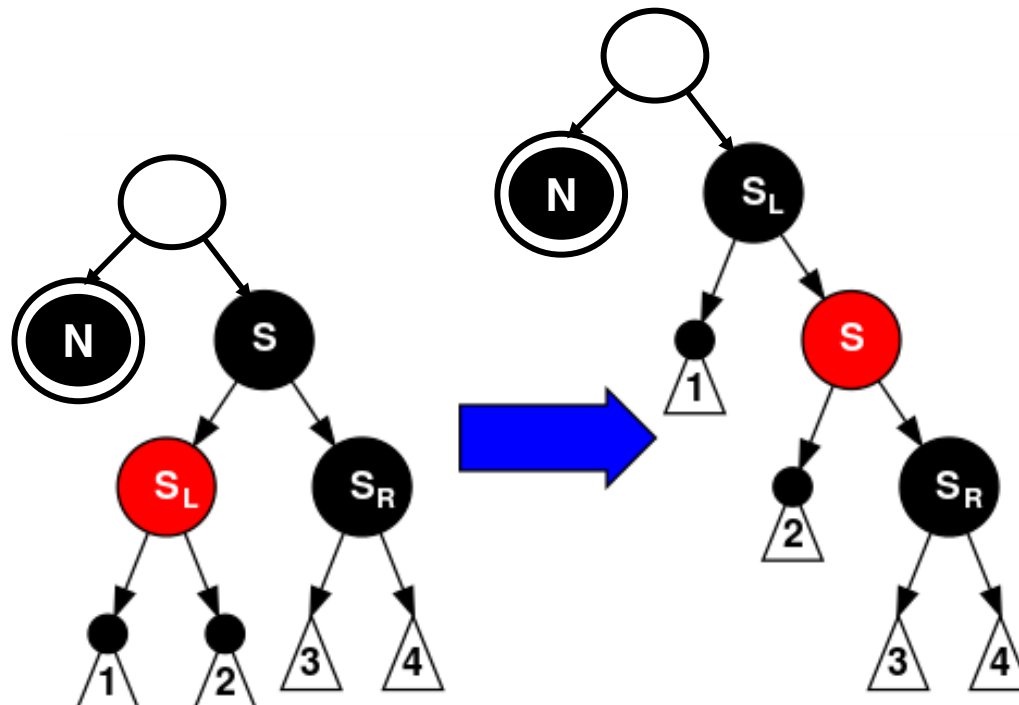
## Case 3-5

- Sibling  $S$  is black,  $S$ 's close child  $S_L$  is red, and a distant child  $S_R$  is black.
- Rotate  $S_L$  in opposite direction to  $N$ .
- Exchange colors between  $S_L$  and  $S$ .



## Case 3-5

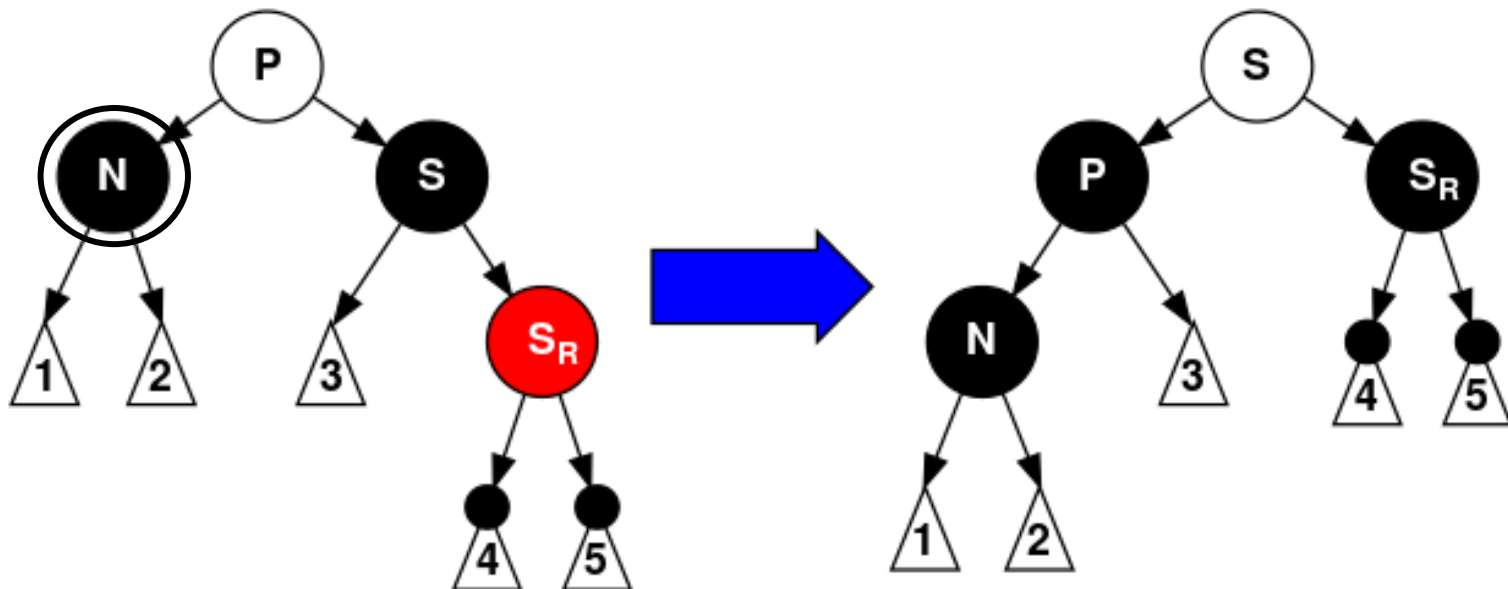
- All paths still have the same number of black nodes, but now **N** has a black sibling whose right child is red, so we fall into case 6.
- Neither **N** nor its parent are affected by this transformation. (For case 6, we relabel **N**'s new sibling as **S**.)





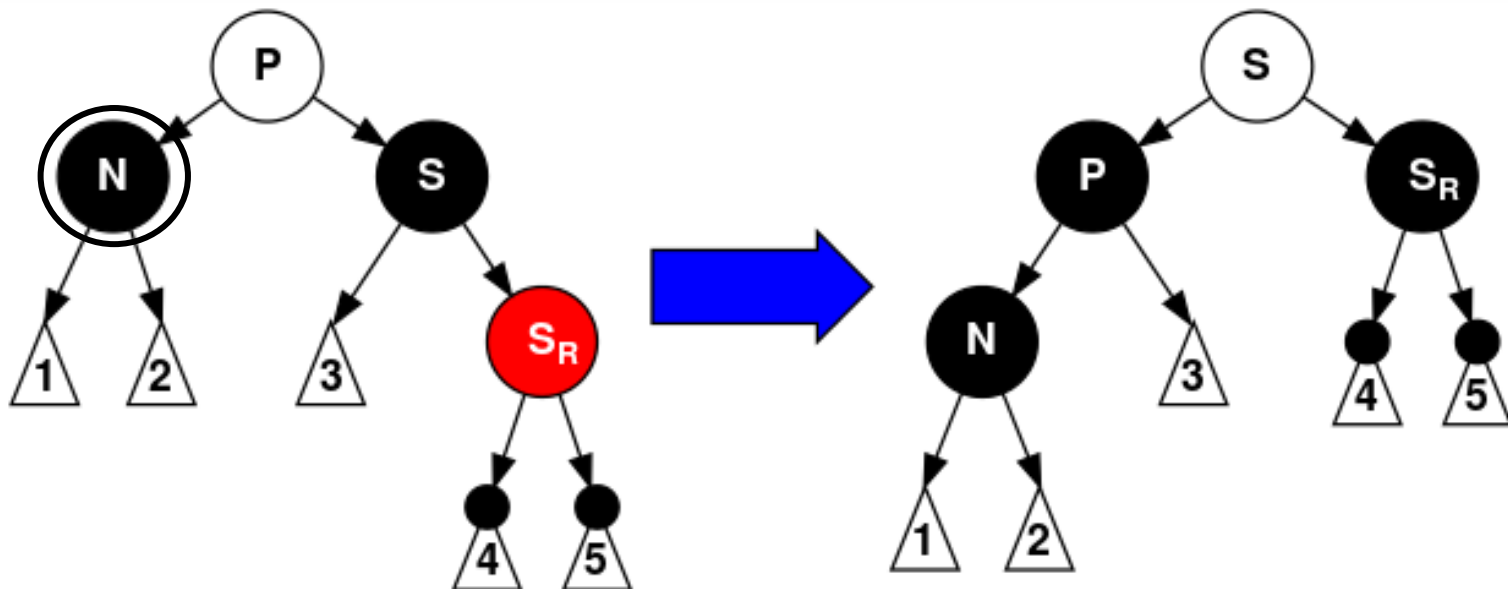
## Case 3-6

- Sibling **S** is black, **S**'s close child **S<sub>L</sub>** is black, and a distant child **S<sub>R</sub>** is red.
- Rotate **P** (either color) toward **N**.
- Exchange colors between **P** and **S**.
- Recolor **S<sub>R</sub>** black.

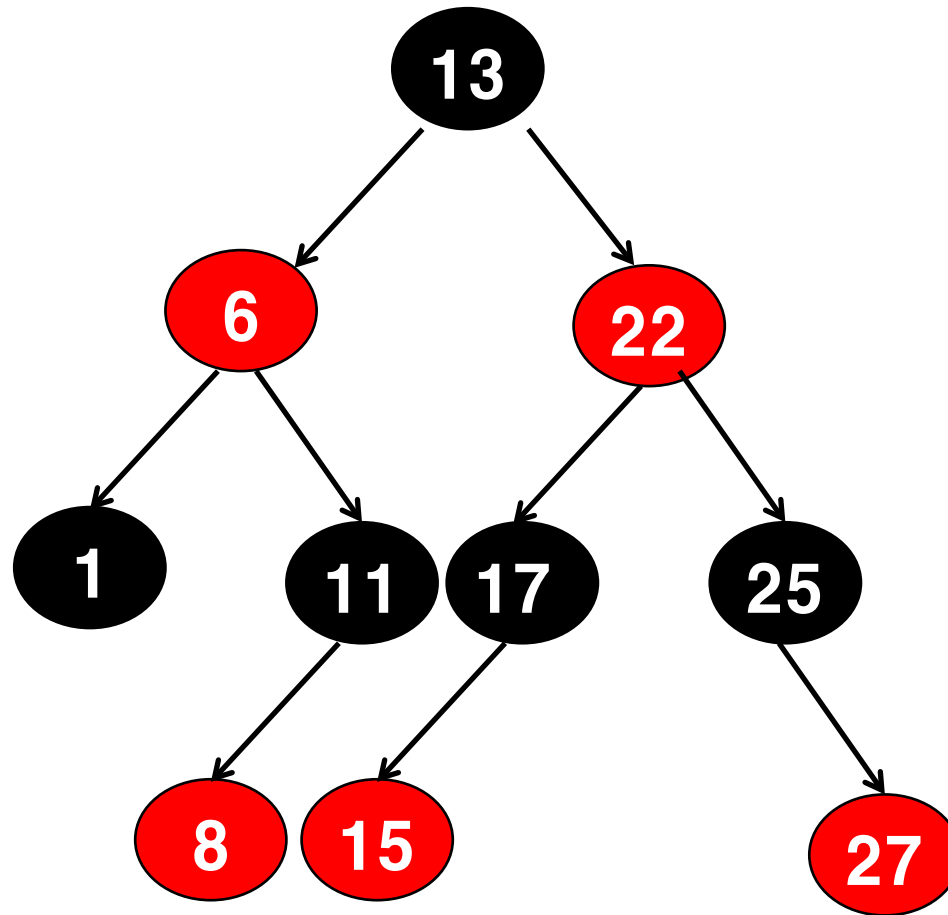


## Case 3-6 (contd.)

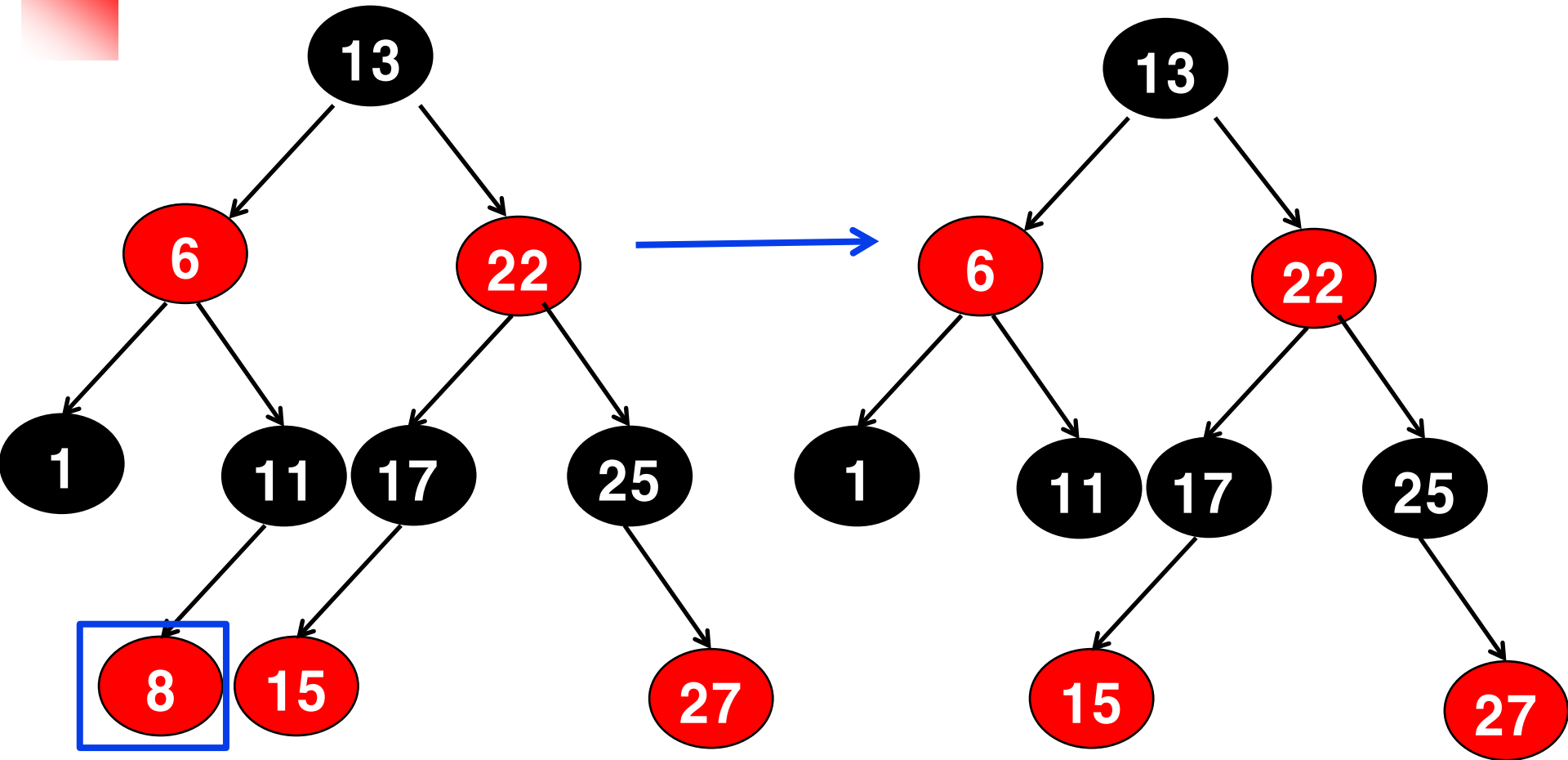
- **N** now has one additional black ancestor: either **P** has become black, or it was black and **S** was added as a black grandparent. Thus, the paths passing through **N** pass through one additional black node.
- The number of black nodes on both paths does not change. Thus, we have restored Depth Property.



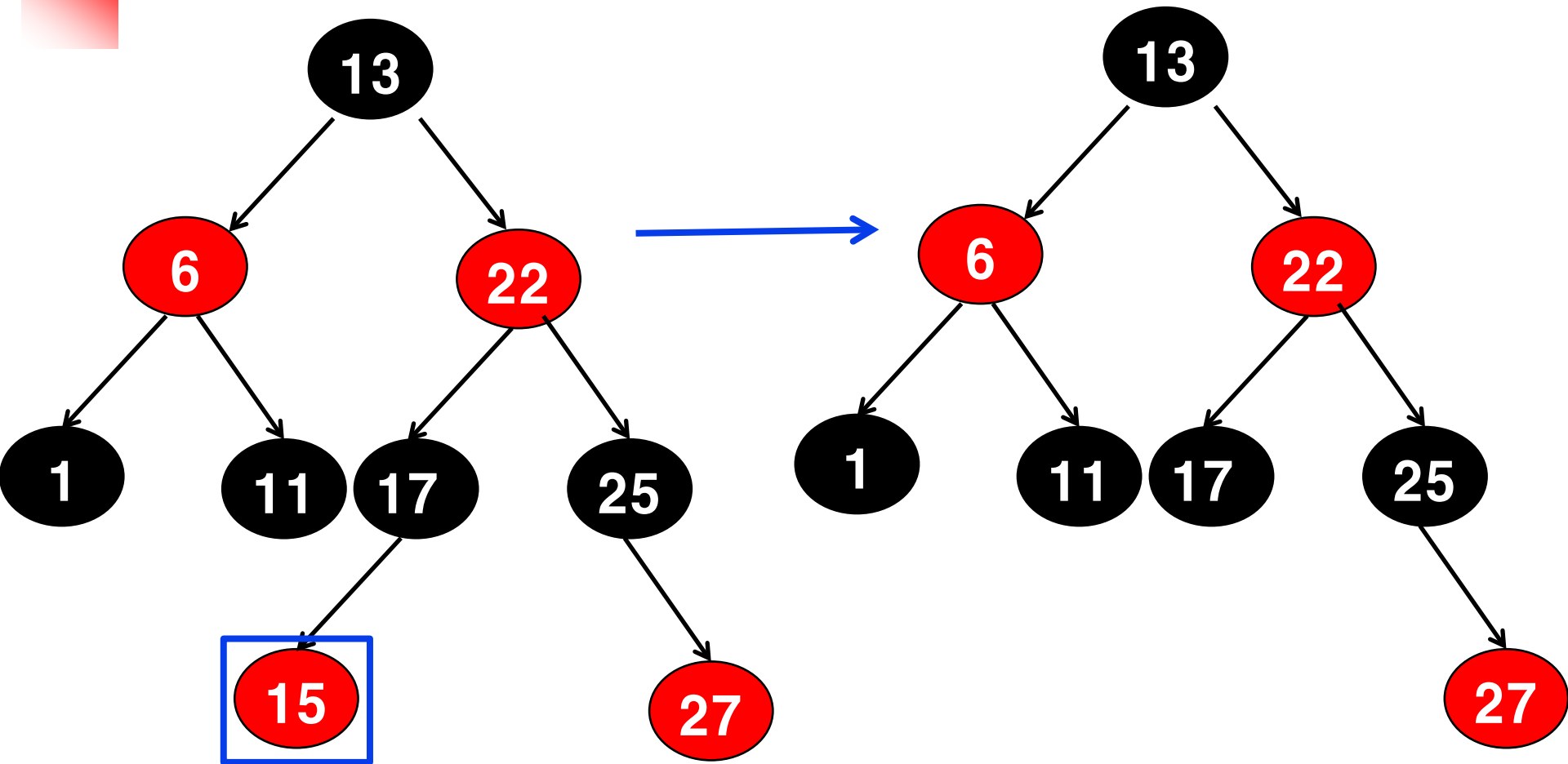
## Exercise: Delete 8, 15, 6, 13, 17, 22



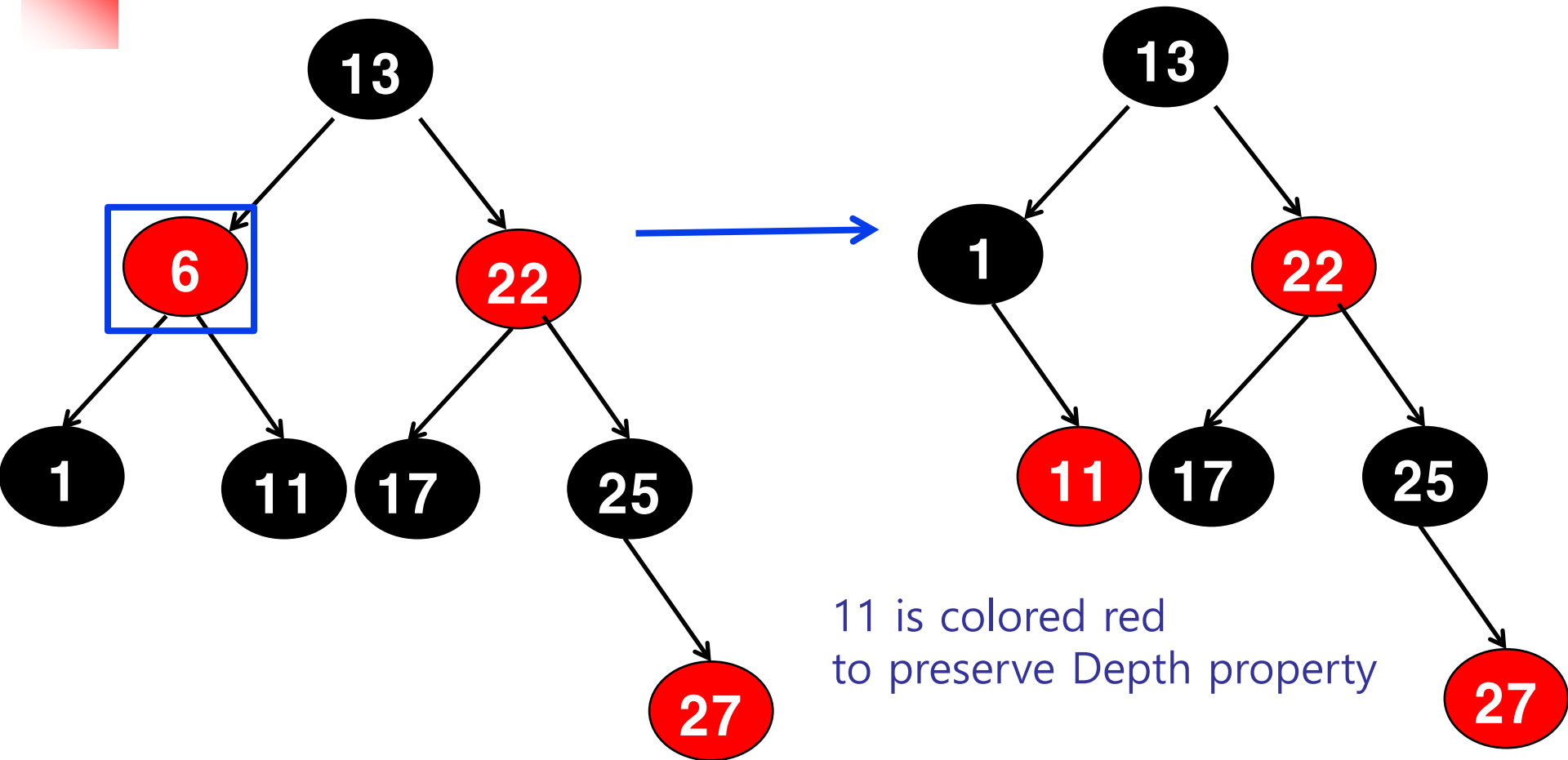
# Solution: Delete 8 (case-1)



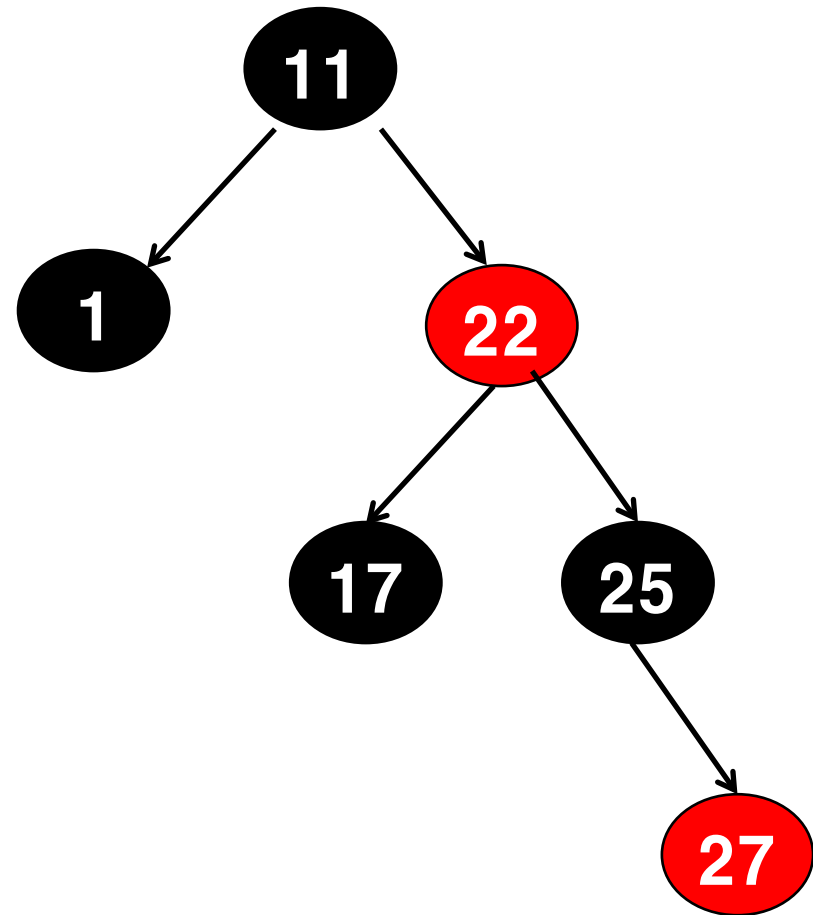
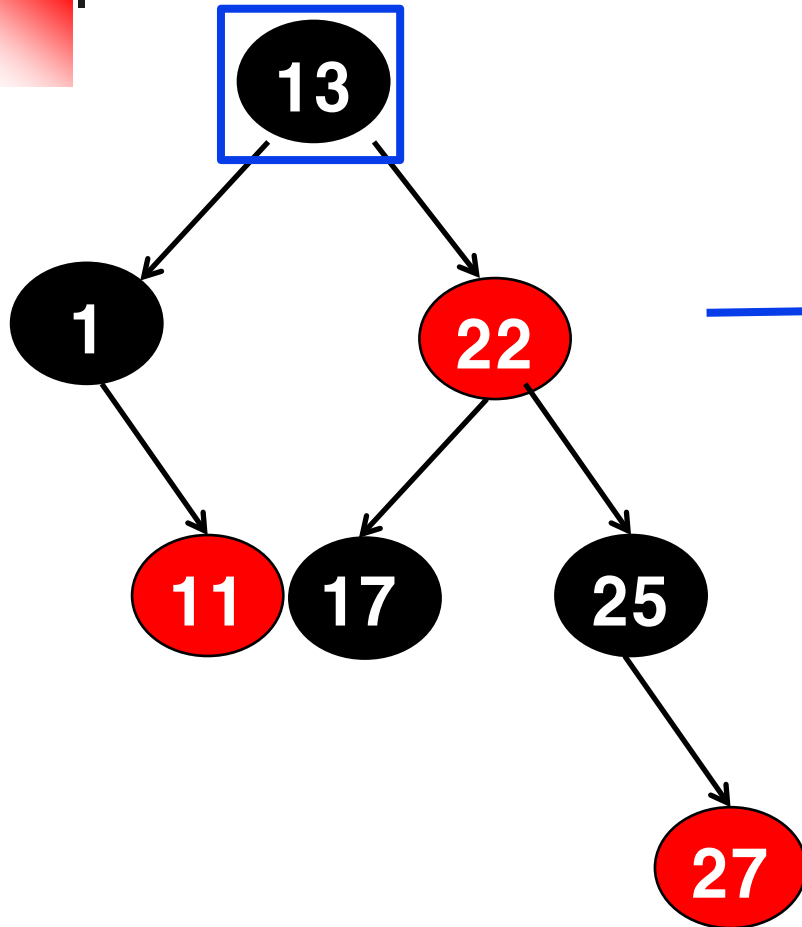
# Solution: Delete 15 (case 1)



# Solution: Delete 6 (case 1)

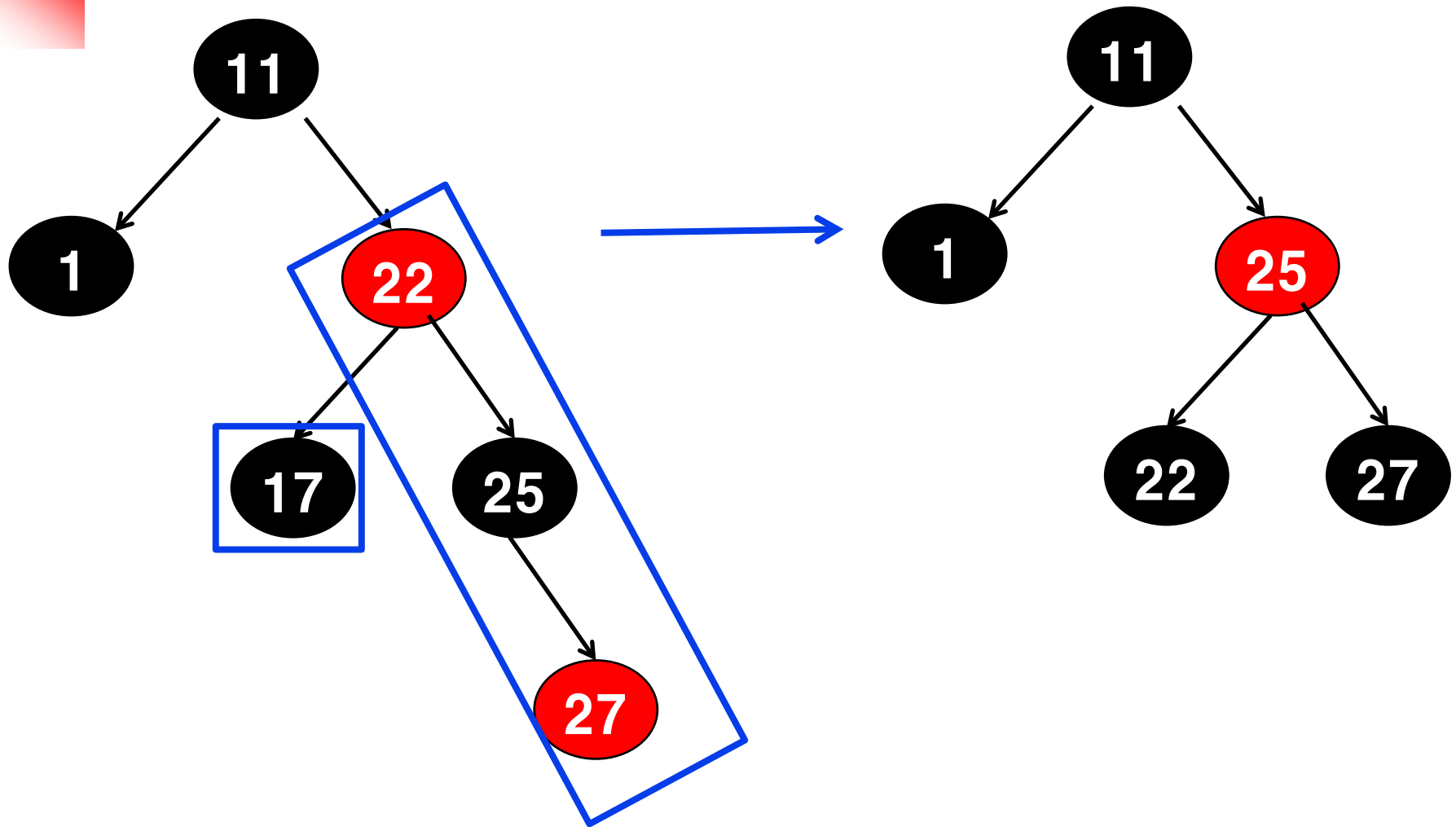


# Solution: Delete 13 (case 1)



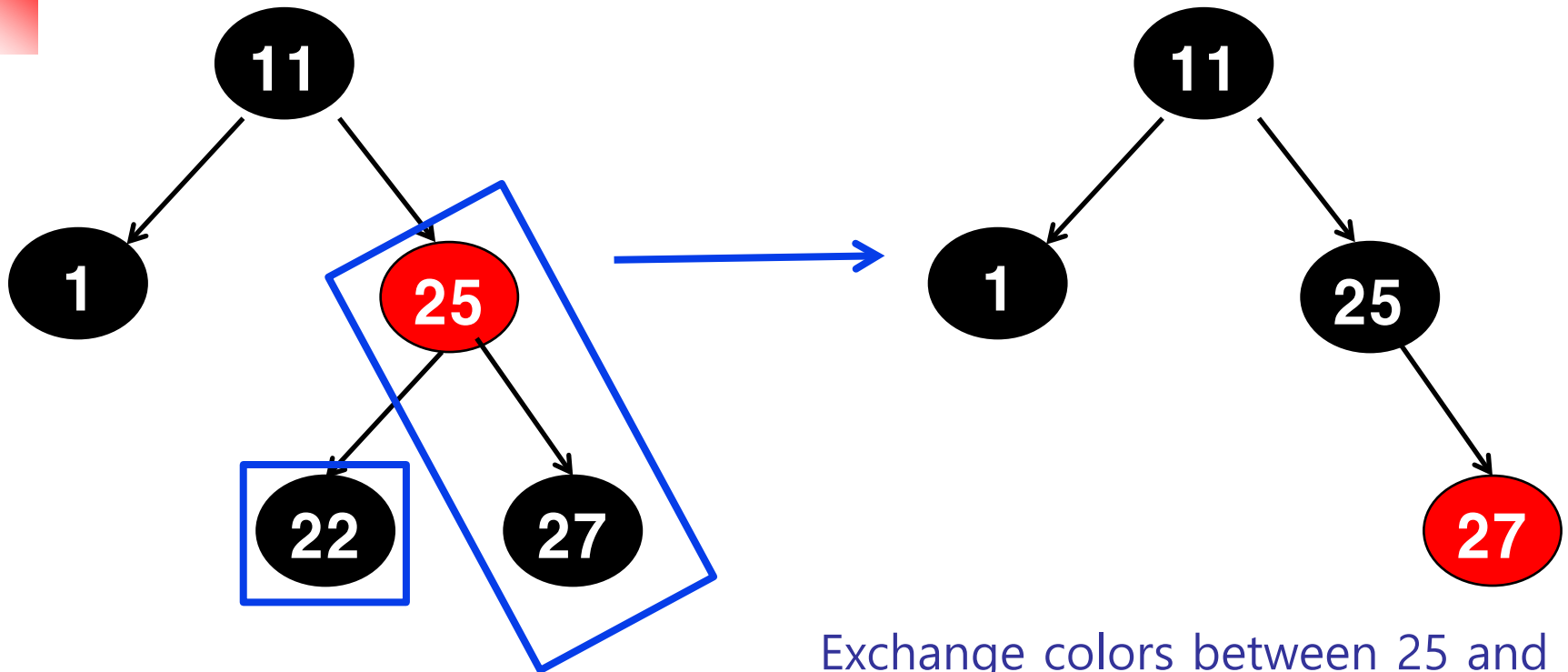
# Solution: Delete 17

(medium case, rotation, p. 58)





## Solution: Delete 22



Exchange colors between 25 and 27  
to preserve Red property



# AVL Tree vs. Red-Black Tree

---

## ■ Similarities

- Both are binary search trees.
- Search, insert, delete follow the same rules.
- Both are balanced trees, but not perfectly balanced.
- Both use tree rotations to rebalance the tree.

## ■ Differences

- AVL Tree uses balance factor to determine balance, and there are 4 cases that require rebalancing.
- Red-Black Tree uses the topology of nearby nodes and their colors to determine balance, and there are many cases that require rebalancing.



---

# Assignment 8

## HW 8

# Red-Black Tree Insert and Delete

---

- Insert

- Cat Dog Bat Fish Chicken Cow Tiger Eagle  
Lion Snake Bird Owl Mouse

- Delete

- Cow Snake Owl Cat Mouse Eagle Bird



# End of Lecture

---