Data Structures:
Red-Black Trees

Won Kim

(Lecture by Youngmin Oh)

Spring 2022

References

- http://en.wikipedia.org/wiki/Red%E2%80%93black_tree
- https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap13c.pdf
- www.cs.unc.edu/~plaisted/comp550/16-rb**trees**.ppt
- http://wwwmayr.informatik.tu-muenchen.de/lehre/2011WS/ea/split/sub-Red-Black-Trees-handout.pdf

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 Sedgewick

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)

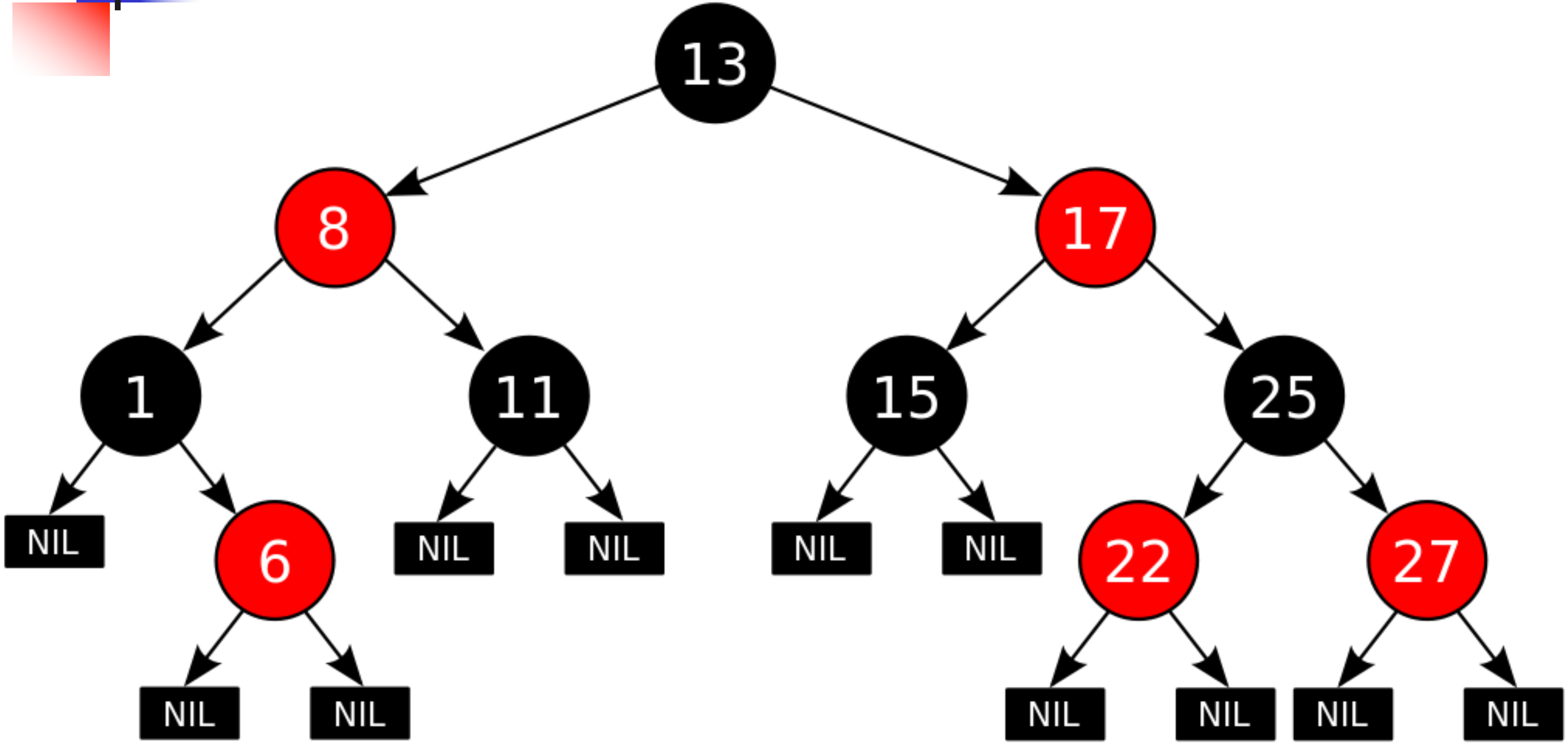
Properties (Rules)

- (1) A node is either red or black.
- (2) The root node is black.
- (3) All leaves (NIL) are black. *(Often omitted for convenience)*
- (4) Every red node must have two black child nodes.
- (5) Every path from a given node to any of its descendant leaves contains the same number of black nodes.

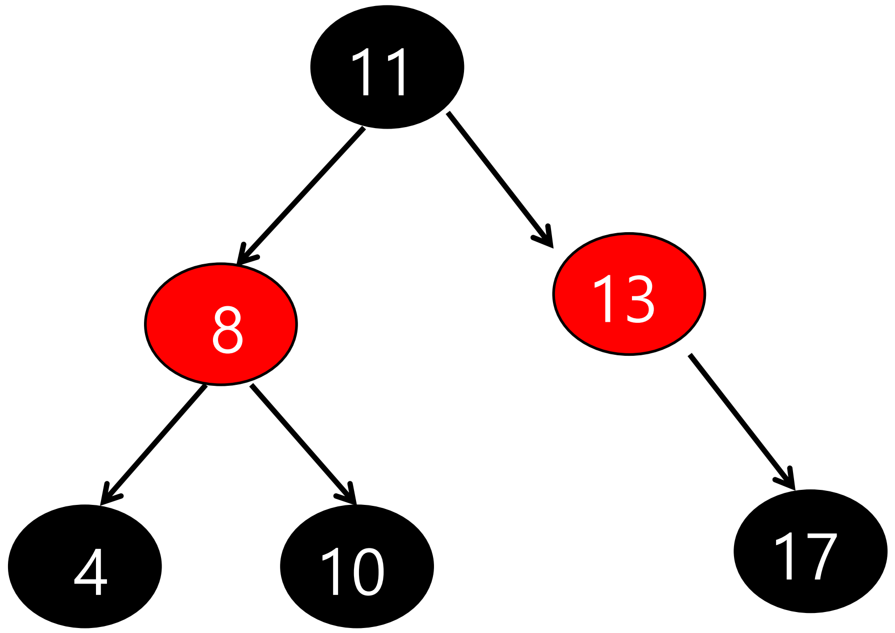
Properties (Rules)

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

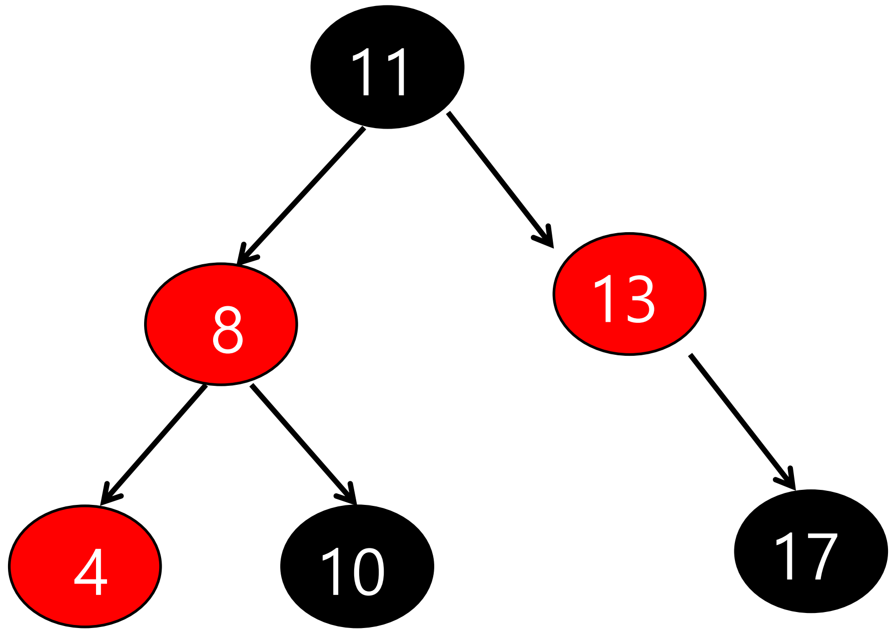
Example Red-Black Tree



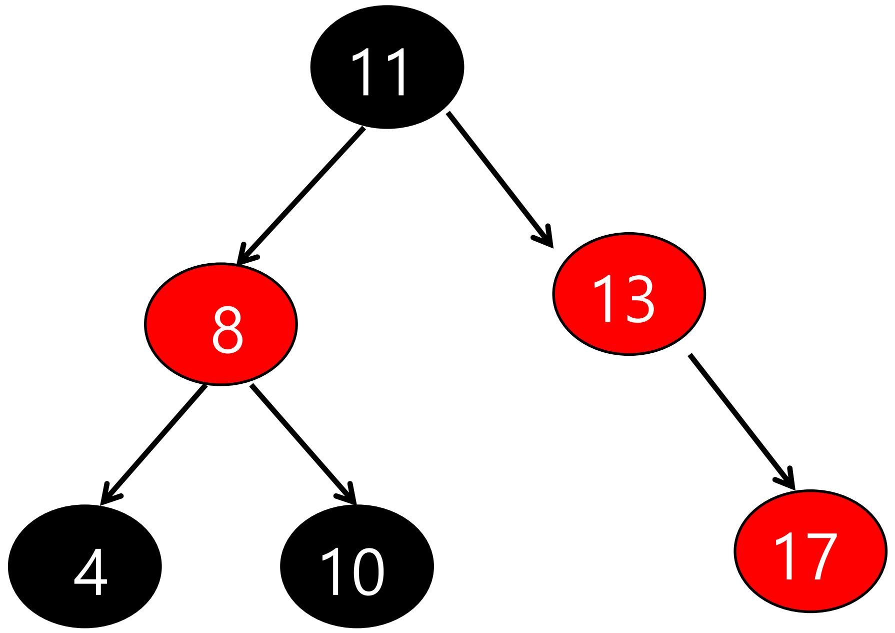
Valid Red-Black Tree? (no)



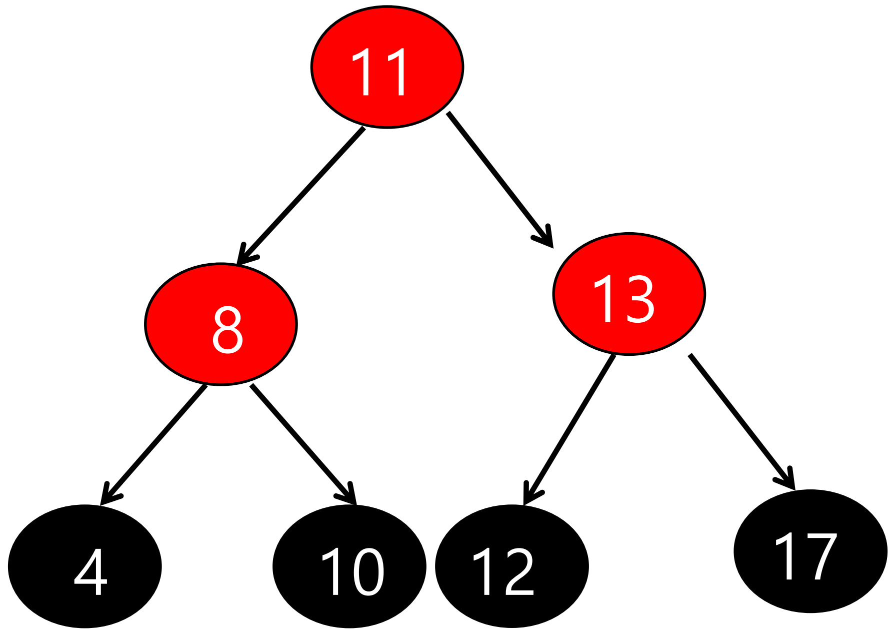
Valid Red-Black Tree? (no)



Valid Red-Black Tree? (no)



Valid Red-Black Tree? (no)

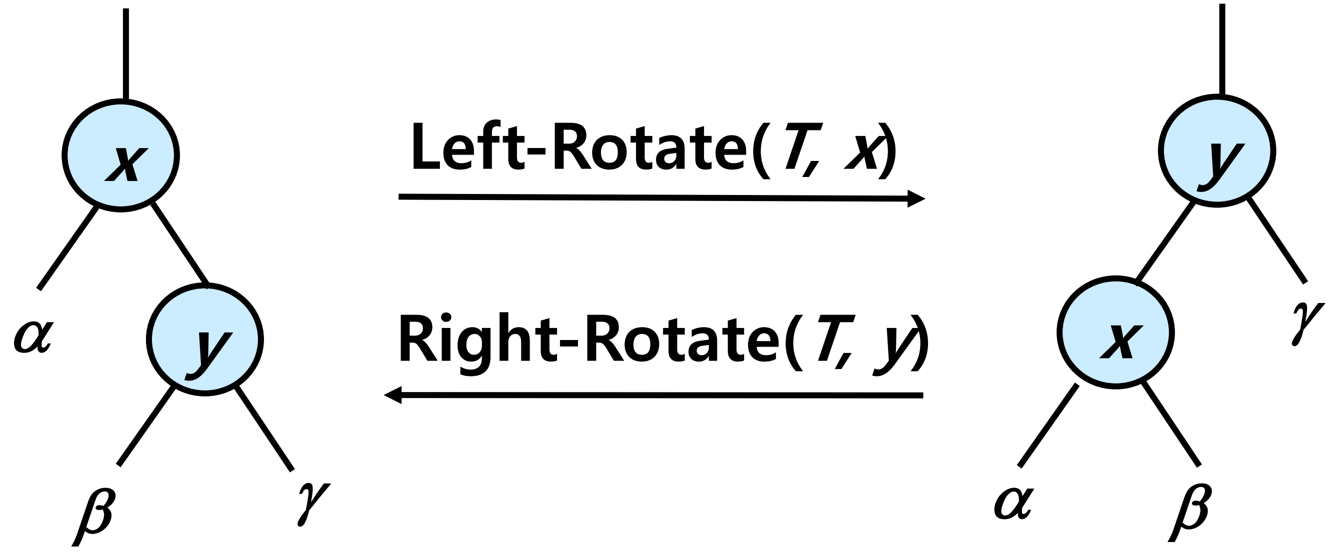


Implementation

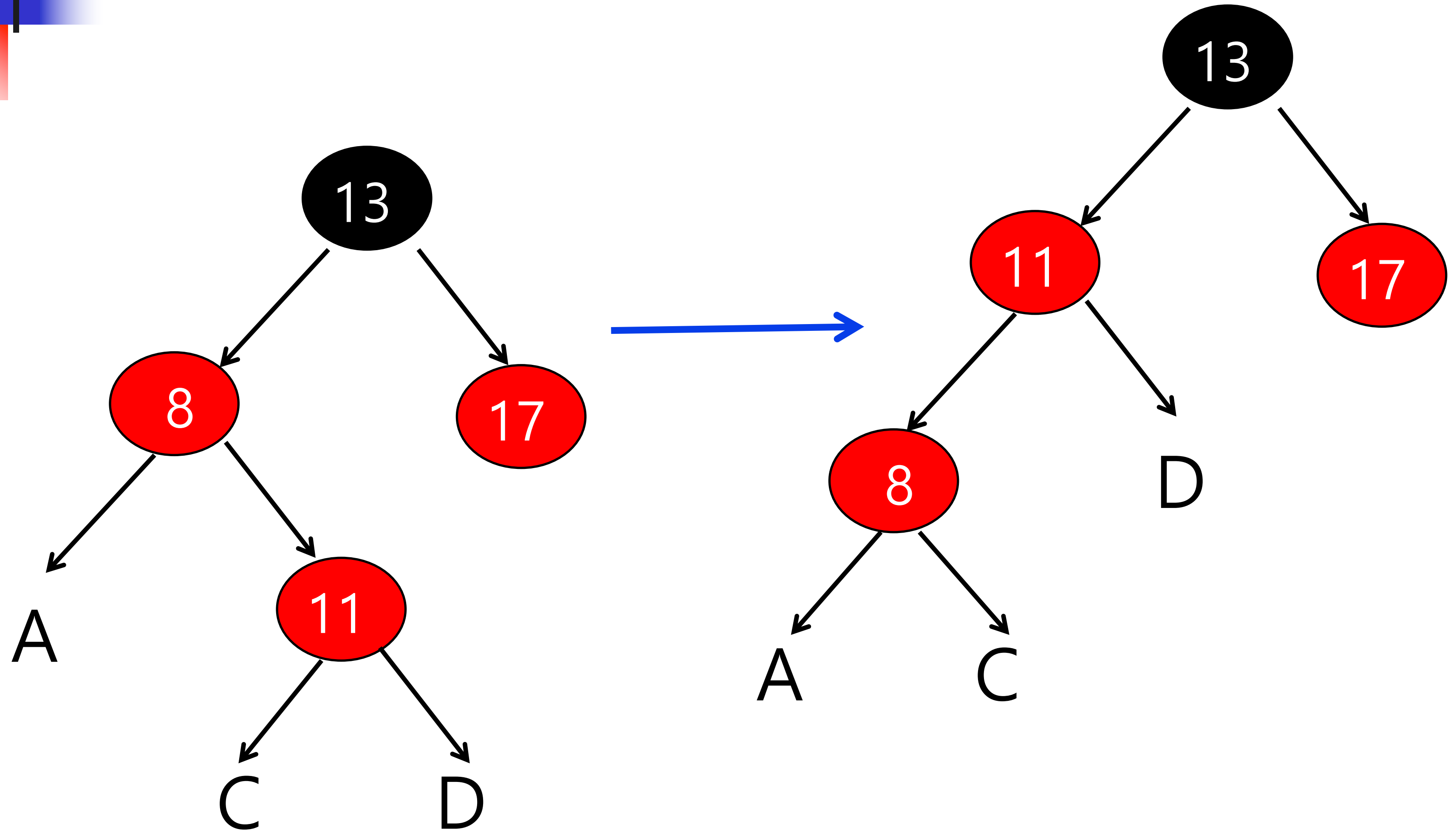
- Binary search tree + 1 bit per node: the attribute *color*, which is either **red** or **black**.
- All other attributes of BSTs are inherited:
  - *key*, *left*, *right*, and *p*.
- All empty trees (leaves) are colored black.
  - We use a single sentinel, *nil*, for all the leaves of red-black tree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

Rotations

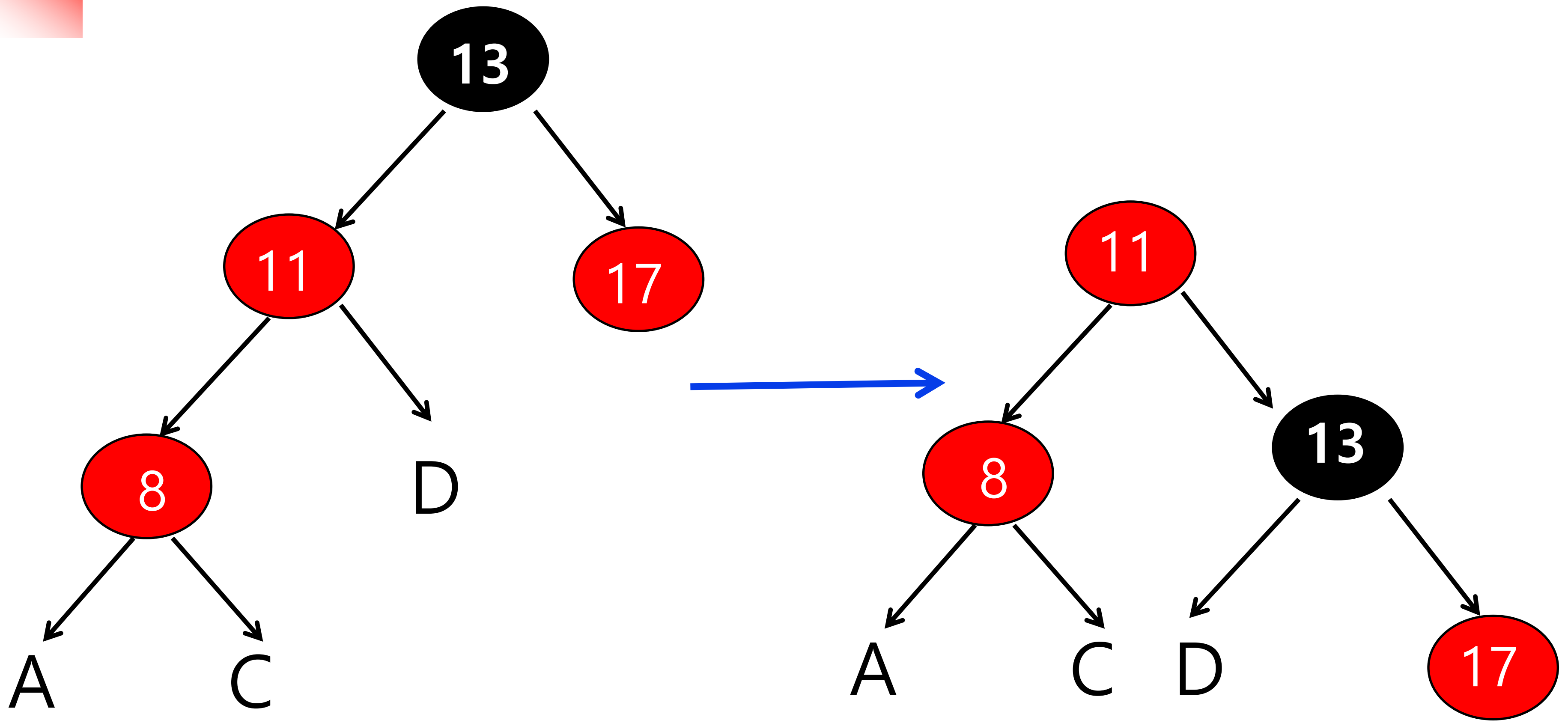
- Rotations are the basic tree-restructuring operation for almost all *balanced* search trees.
- Rotation takes a red-black-tree and a node,
- Changes pointers to change the local structure, and preserve the binary search tree property.
- Left rotation and right rotation are inverses.



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. (property 4)
- 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

- property 3 (all leaves are black) always holds.
- property 4 (both children of every red node are black) is violated only by adding a red node, repainting a black node red, or a rotation.
- property 5 (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)

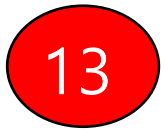
Insertion: 5 Cases

- **(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)

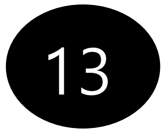
Insertion: Case 1

- The current (new) node **N** is the root of the tree.
- N is repainted black to satisfy property 2.
- Property 5 is not violated: The insertion adds one black node to every path at once.

Example: Insert 13



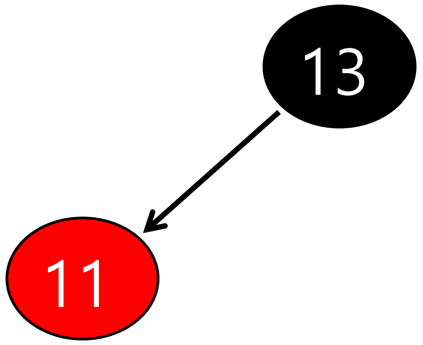
Example: Insert 13



Insertion: Case 2

- The current node's parent **P** is black.
- The tree is still valid. Property 4 is not violated.
- Property 5 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.

Example: insert 11

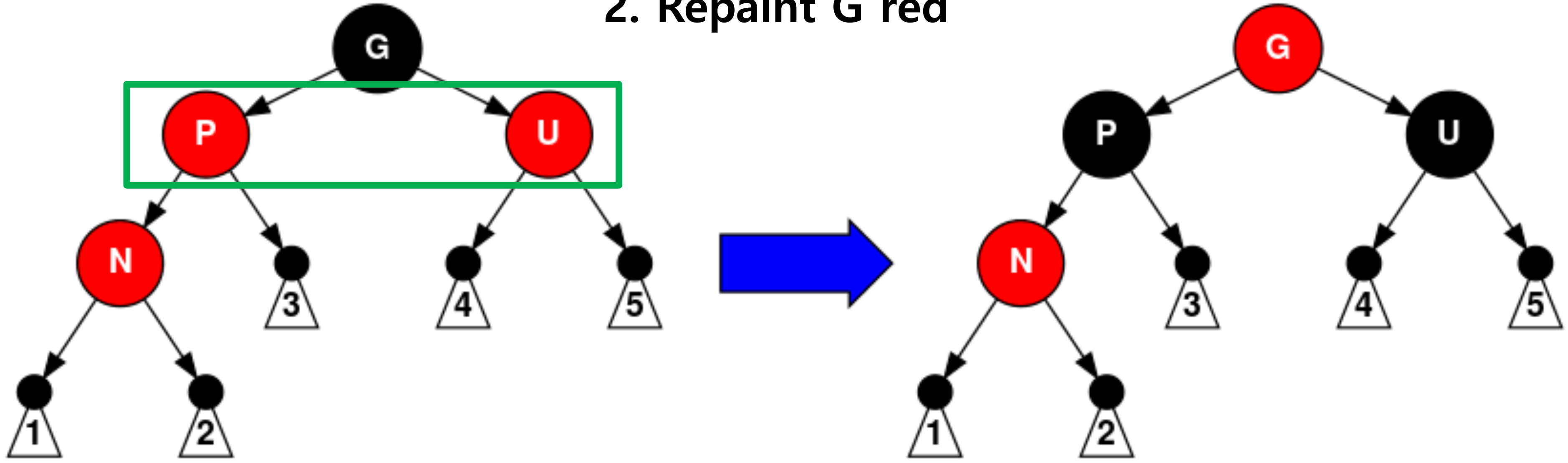


Insertion: Case 3

- If both the parent **P** and the uncle **U** are red, both can be repainted black and the grandparent **G** becomes red (to maintain property 5).
- Property 5 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.
- However, the grandparent **G** may now violate properties 2 or 4 (property 4 possibly being violated since **G** may have a red parent).
- To fix this, the entire procedure is recursively performed on **G** from case 1.

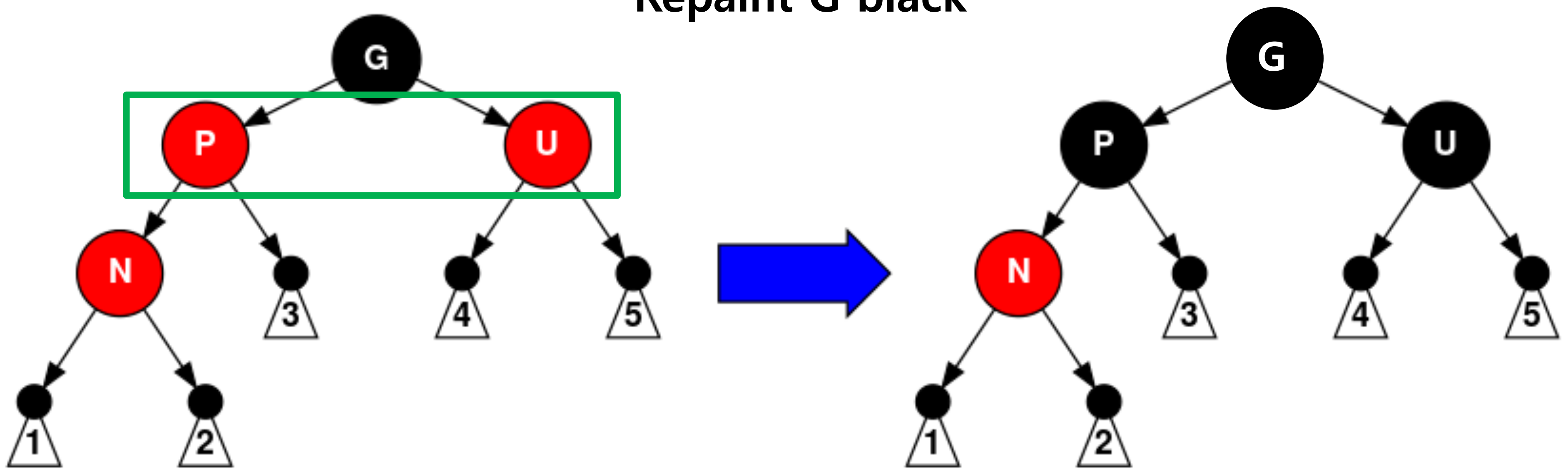
Insertion: Case 3

1. Repaint P and U black
2. Repaint G red



Insertion: Case 1

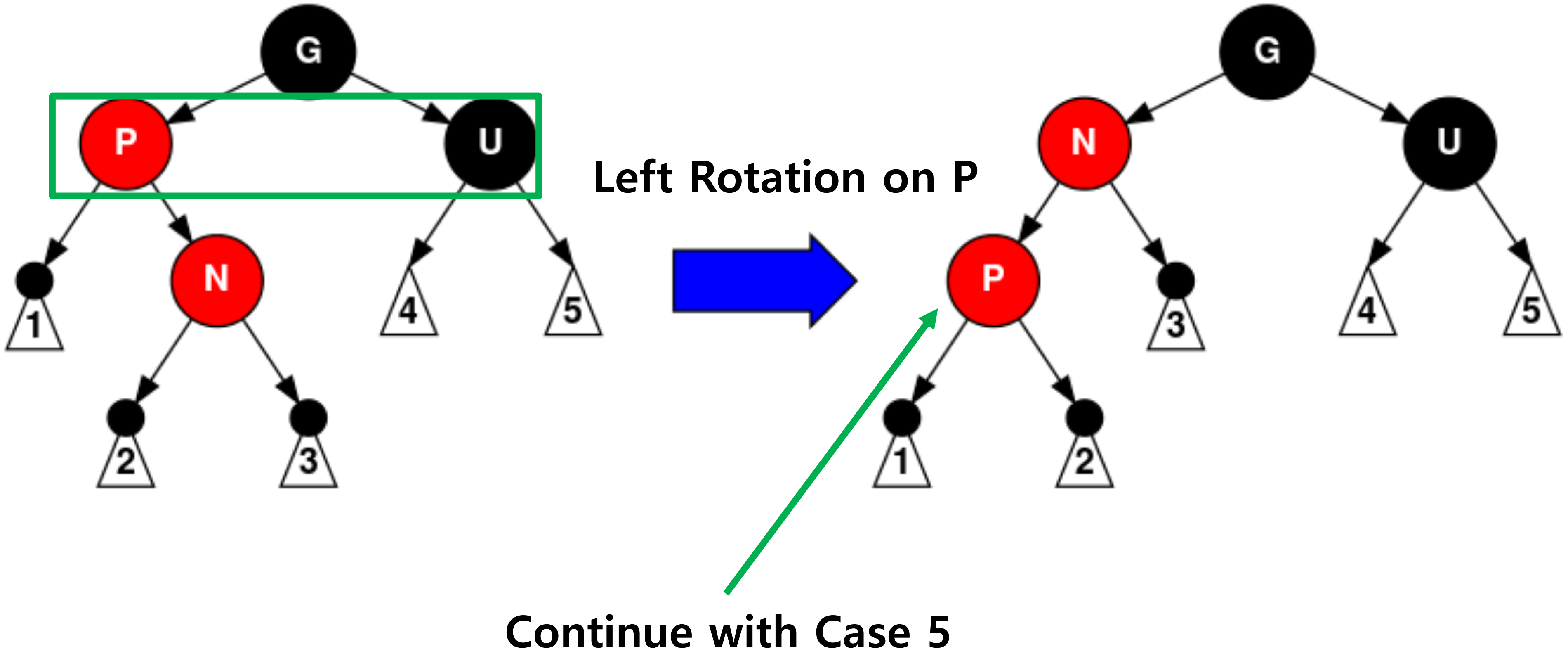
Repaint G black



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.
- Then, the former parent node **P** is dealt with using case 5 (relabeling **N** and **P**) because property 4 is still violated.
- Property 5 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 property 5 is not violated by the rotation.
- After this case has been completed, property 4 is still violated, and this is resolved by continuing to case 5.

Insertion: Case 4

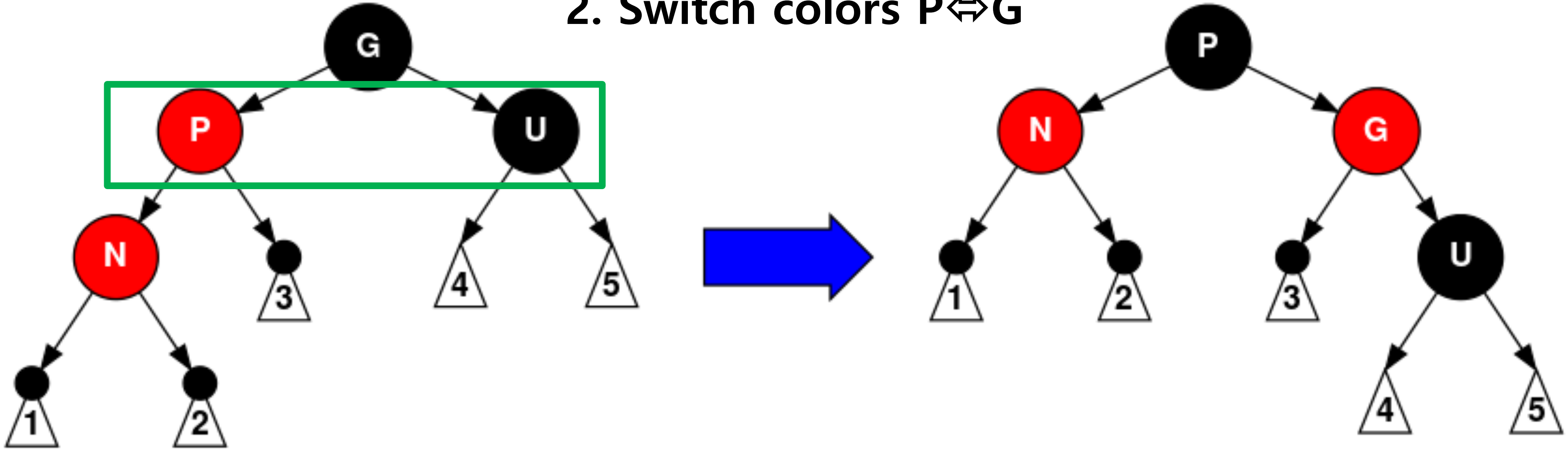


Insertion: Case 5

- 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**.
- **G** is known to be black, since its former child **P** could not have been red otherwise (without violating property 4).
- Then, the colors of **P** and **G** are switched, and the resulting tree satisfies property.
- Property 5 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 5

1. Right Rotation on G
2. Switch colors P⇔G



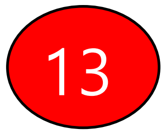
Insertion: Case 4 and 5 mirrored

- The whole process of case 4 and 5 can be mirrored (left <-> right)

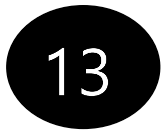
Exercise

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

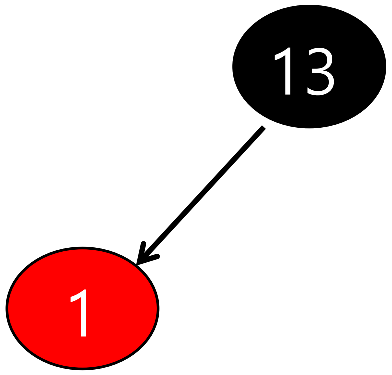
Solution: Insert 13 (case 1)



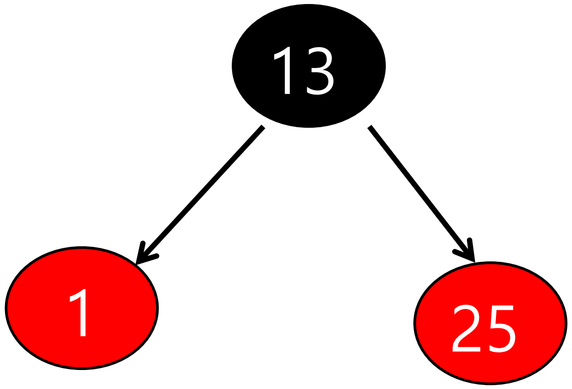
Solution: Insert 13 (case 1)



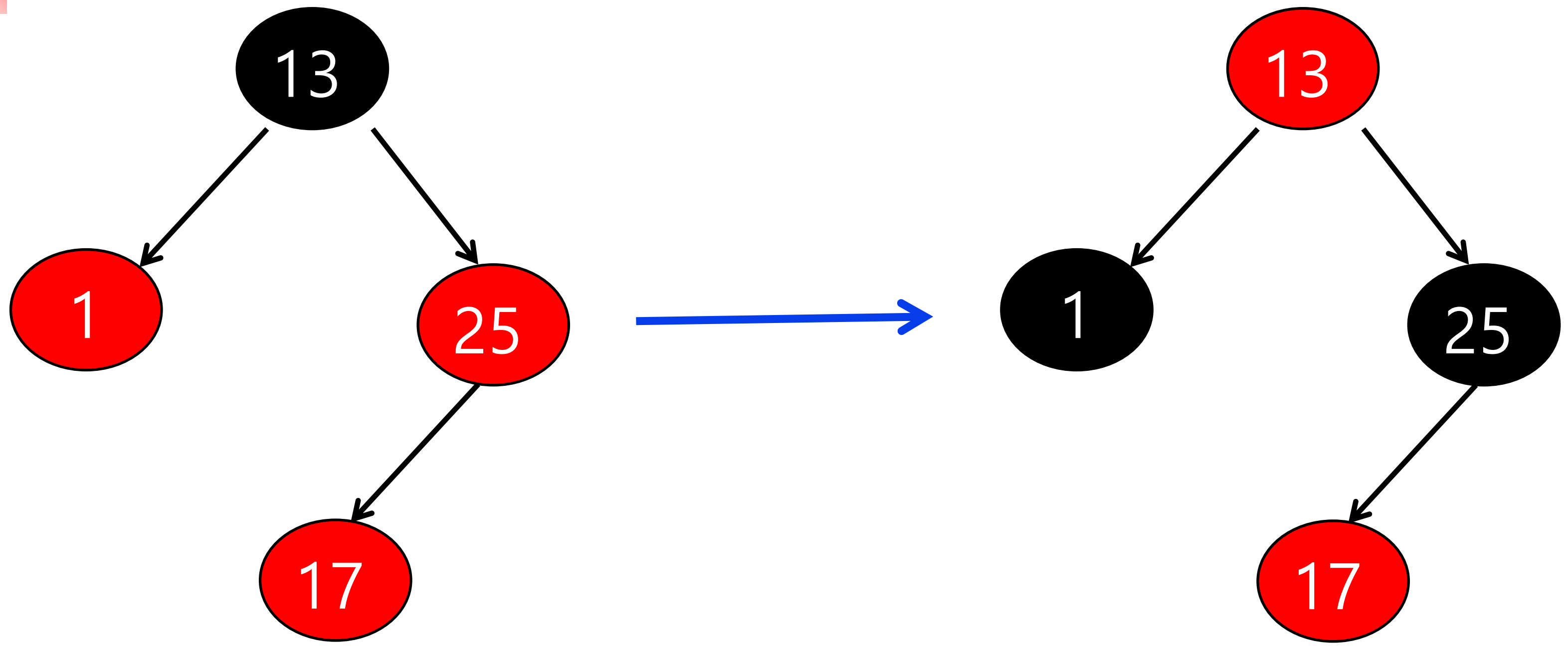
Solution: Insert 1 (case 2)



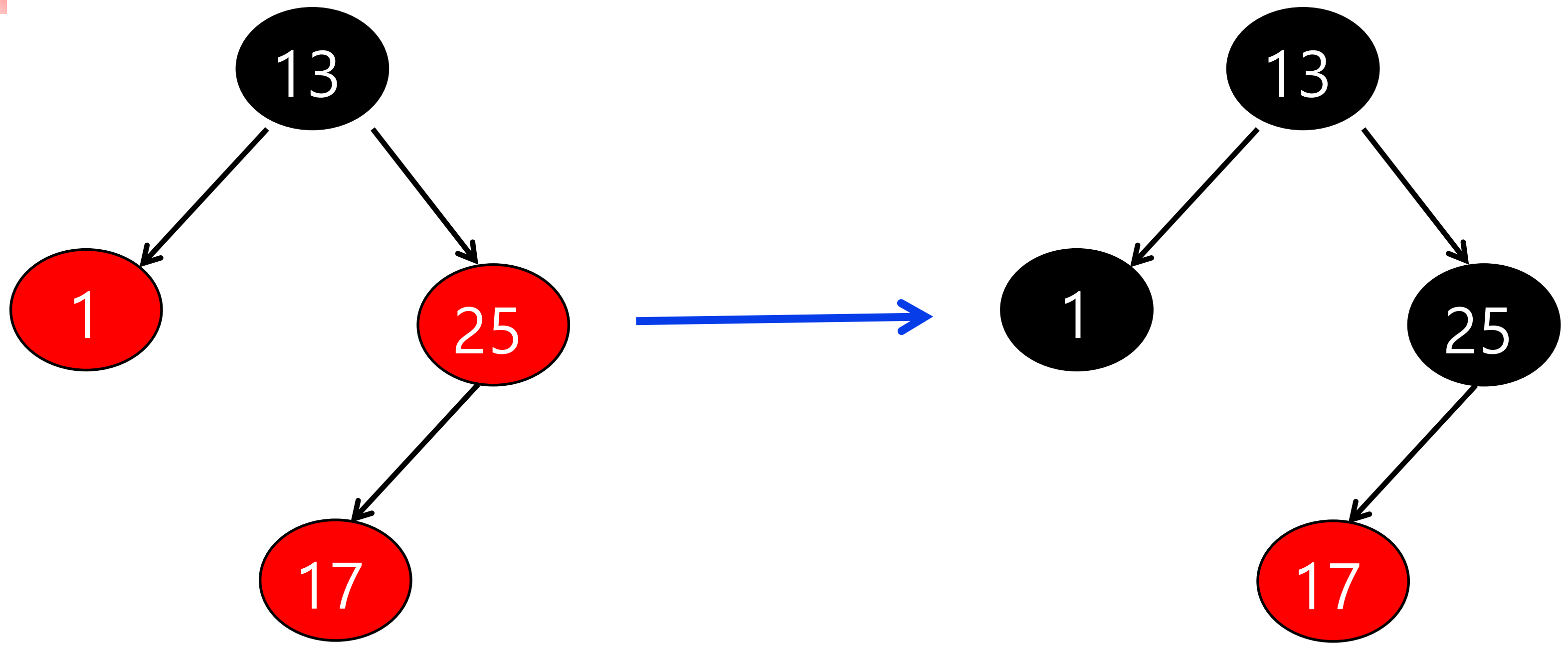
Solution: Insert 25 (case 2)



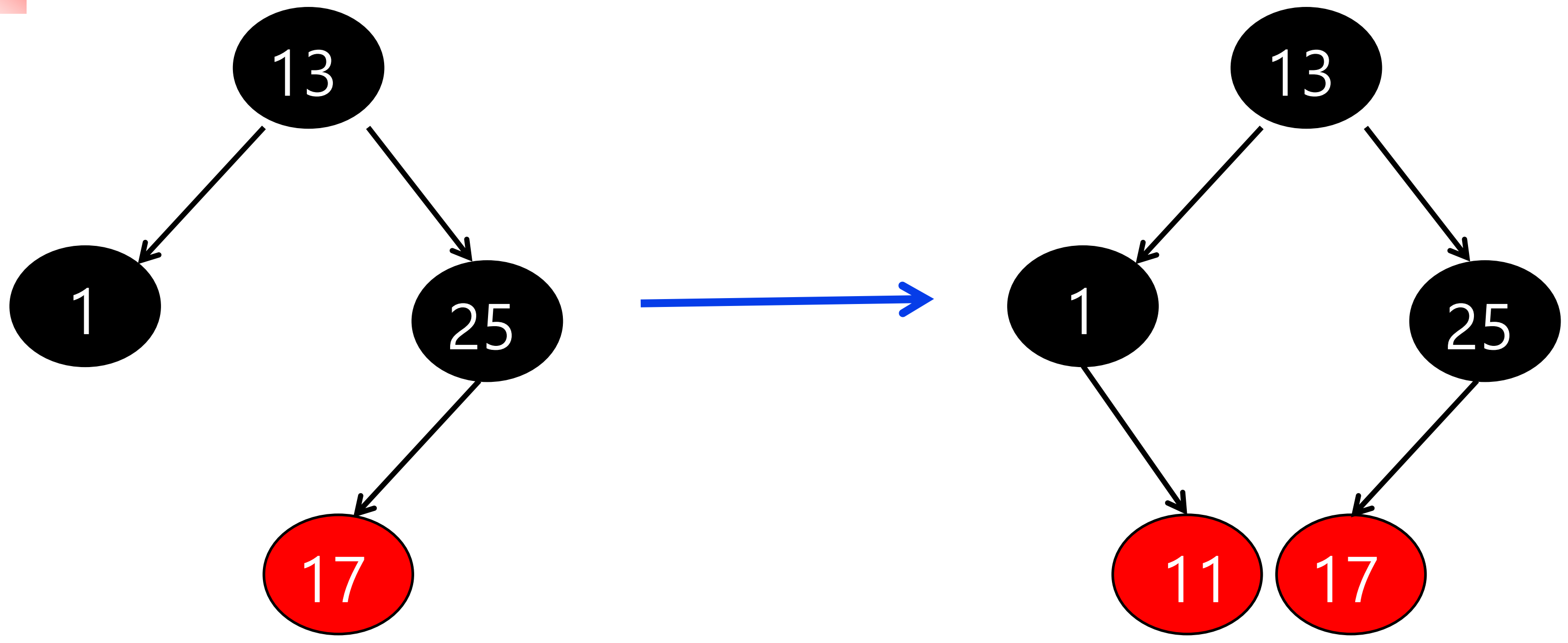
Solution: Insert 17 (case 3)



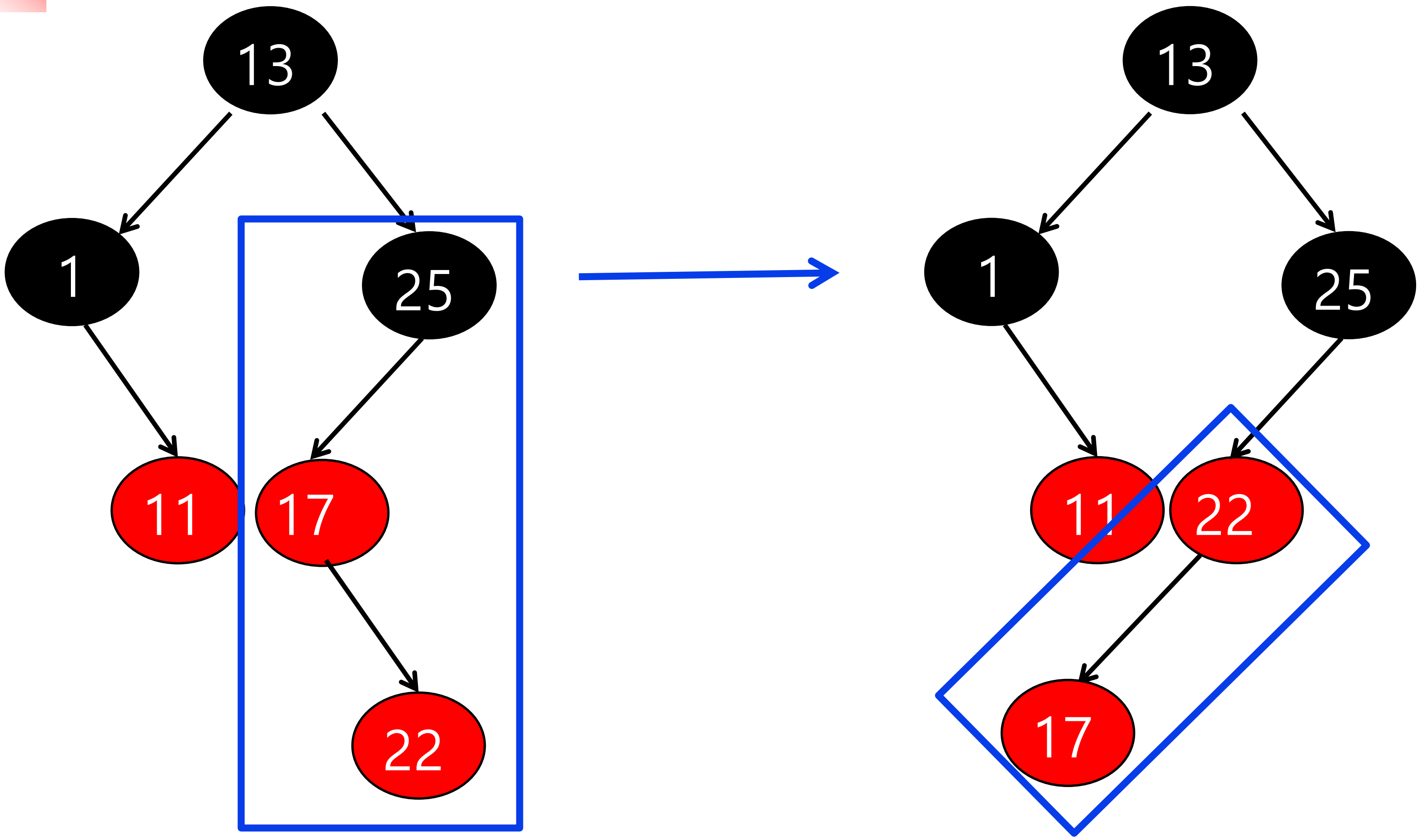
Solution: Insert 17 (case 1)



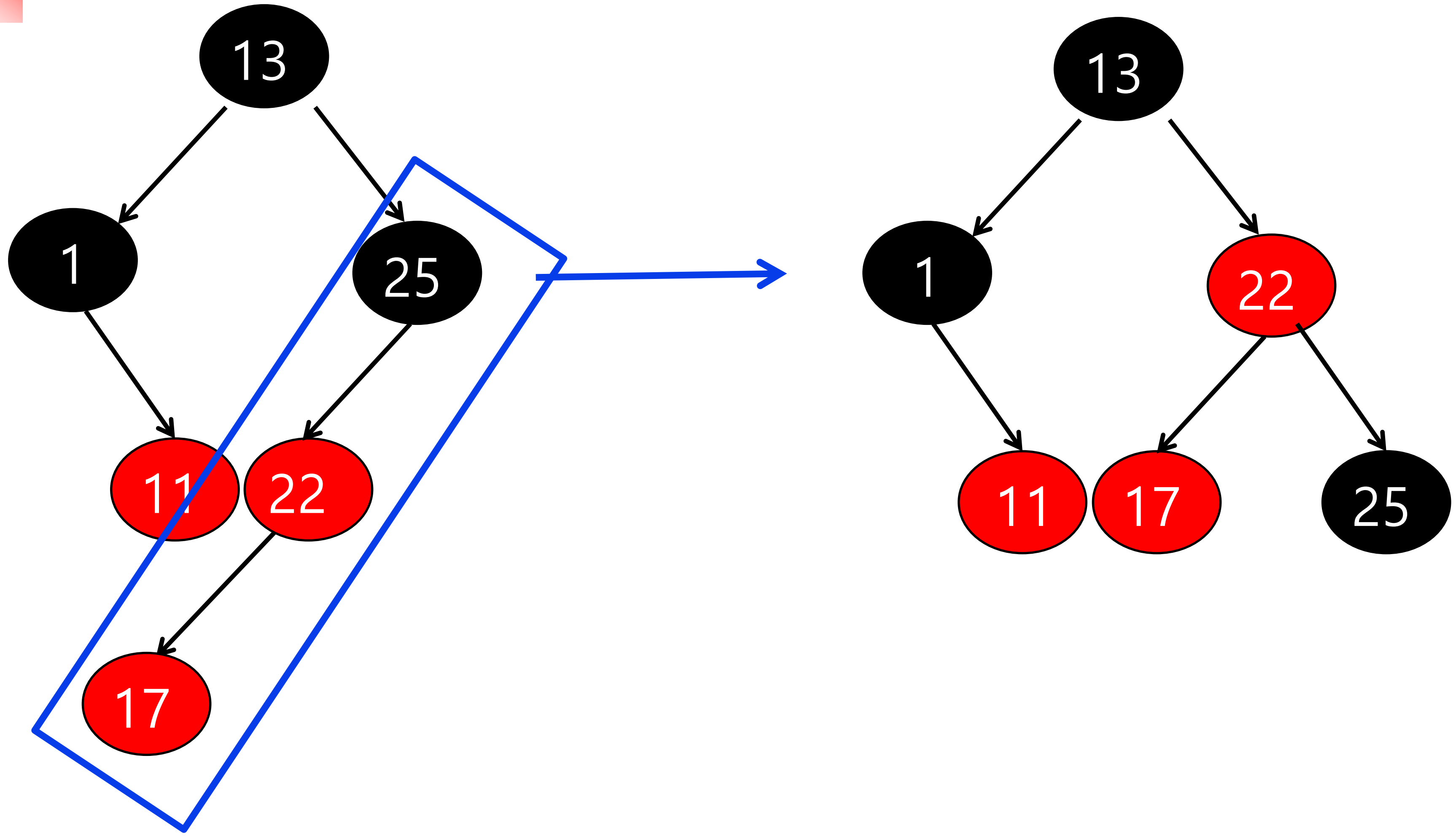
Solution: Insert 11 (case 2)



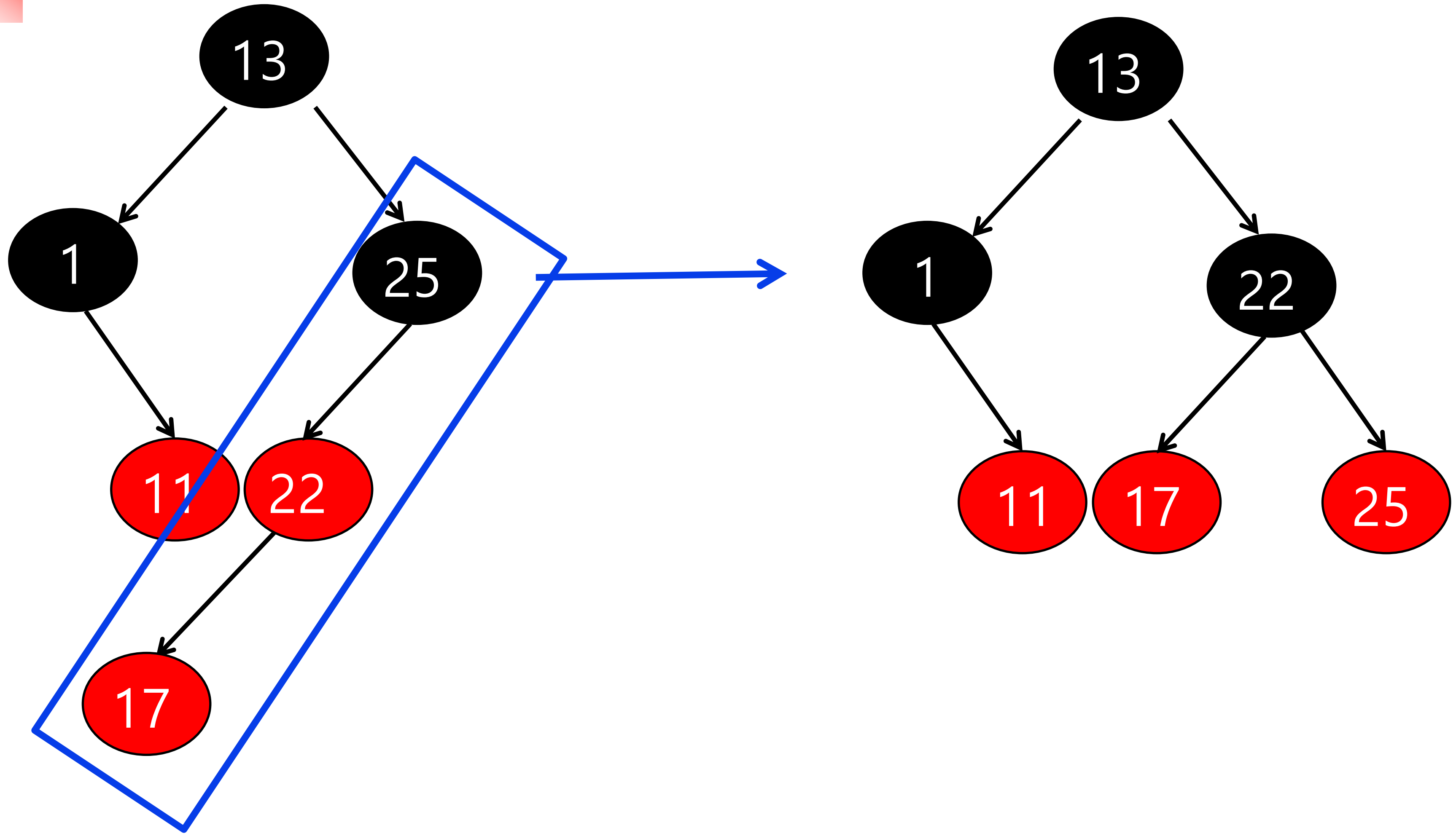
Solution: Insert 22 (2 rotations) (case 4)



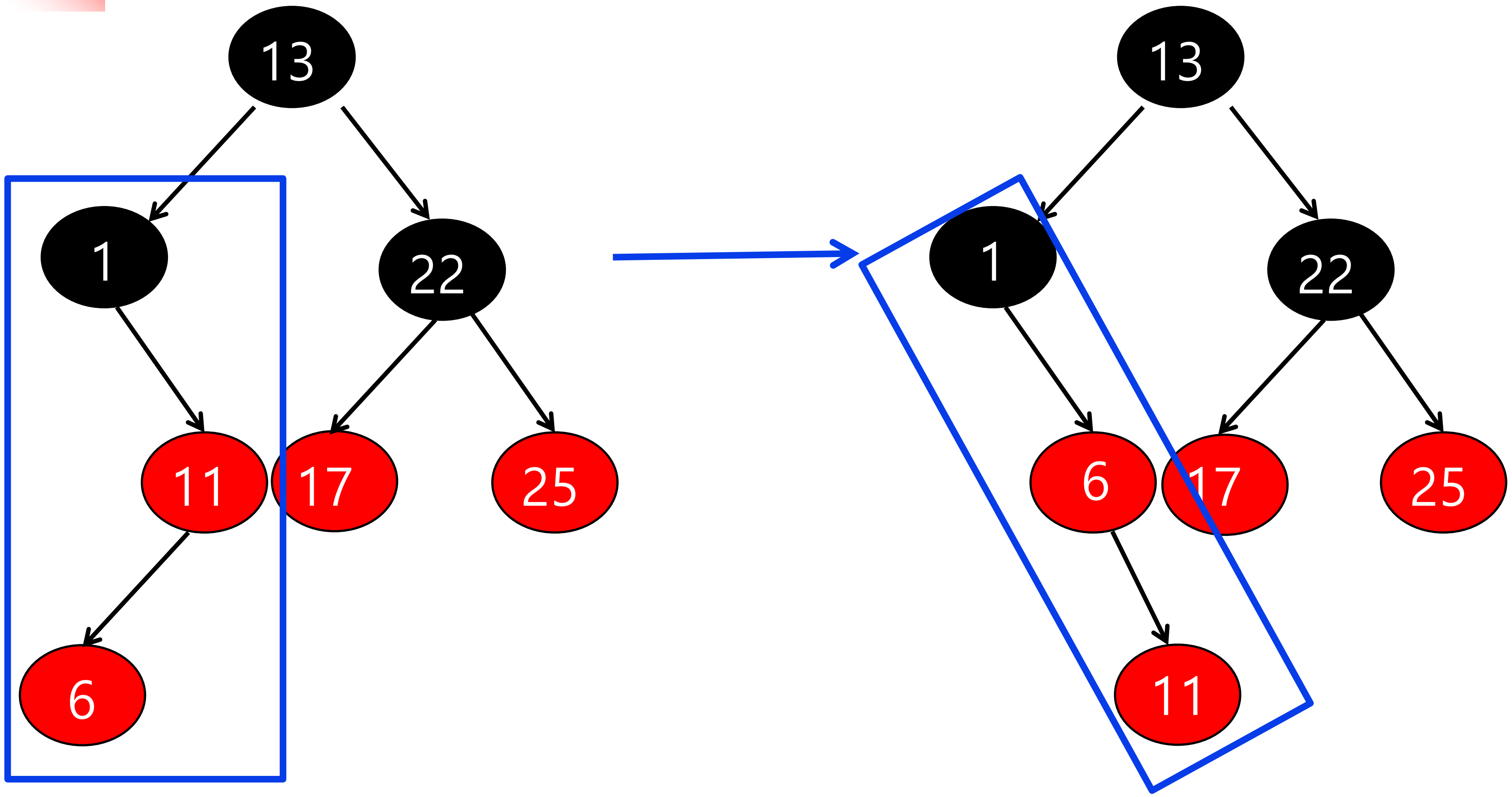
Solution: Insert 22 (2 rotations) (case 5)



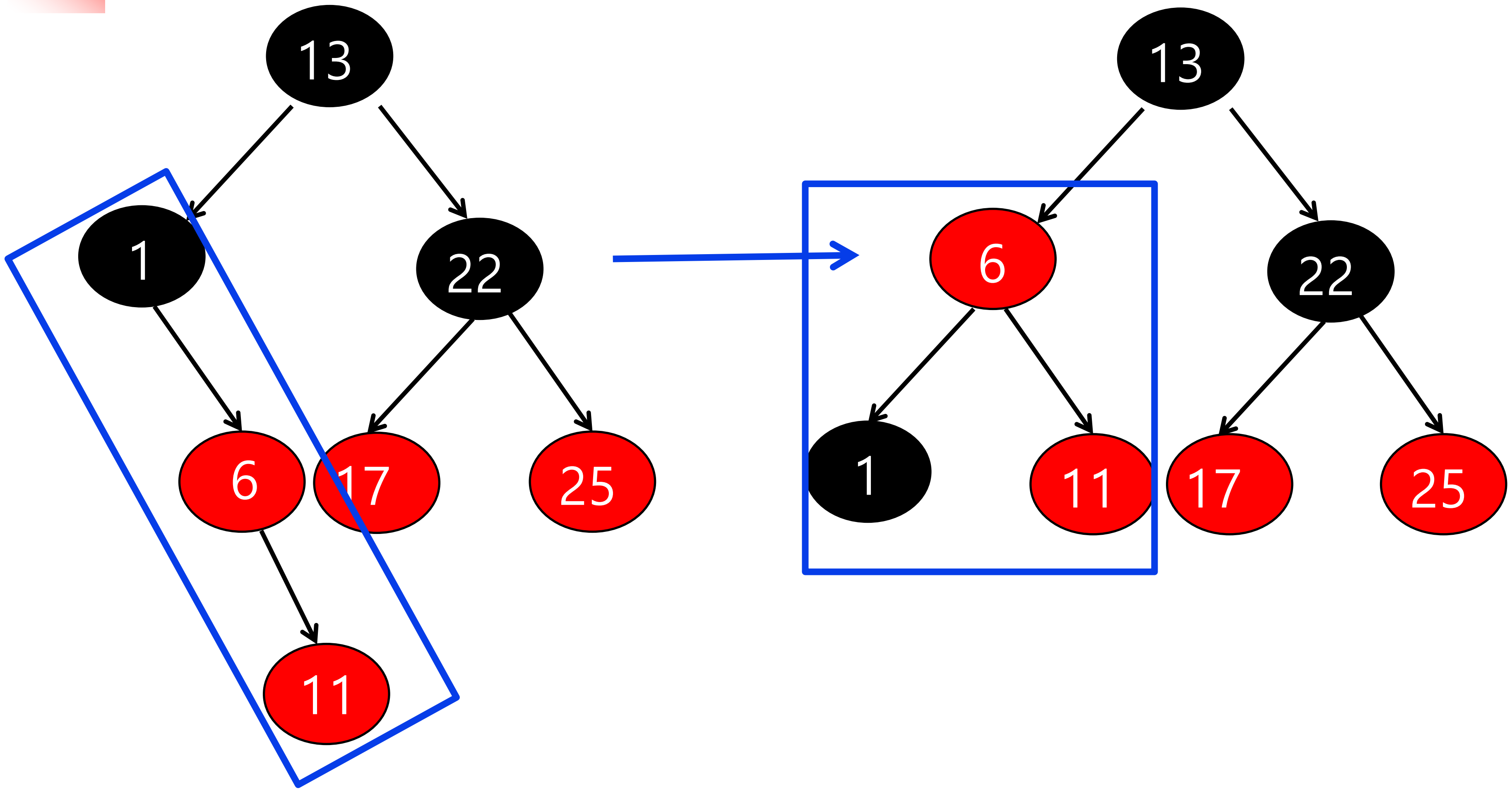
Solution: Insert 22 (2 rotations) (case 5)



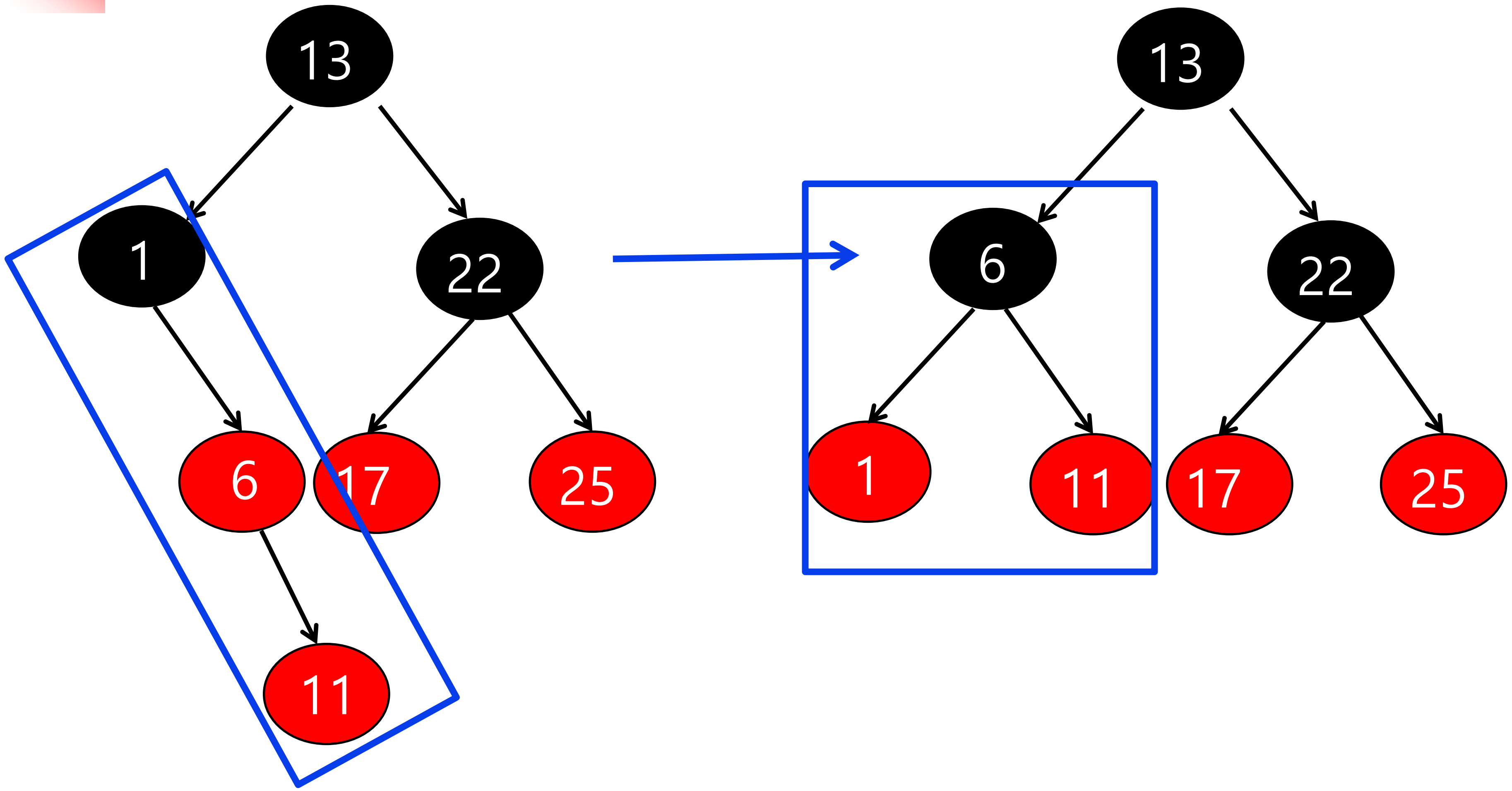
Solution: Insert 6 (2 rotations) (case 4: mirror case)



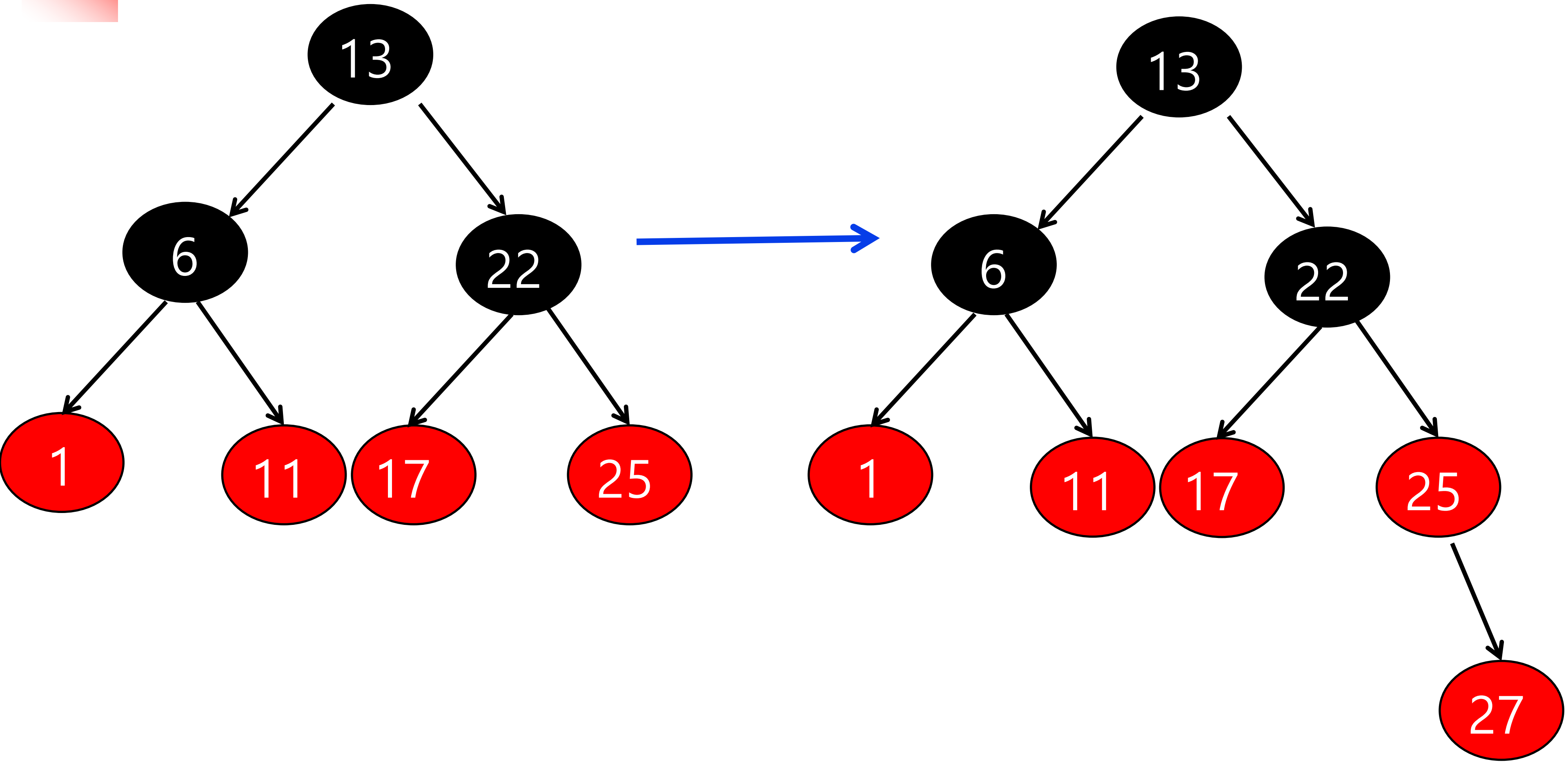
Solution: Insert 6 (2 rotations) (case 5: mirror case)



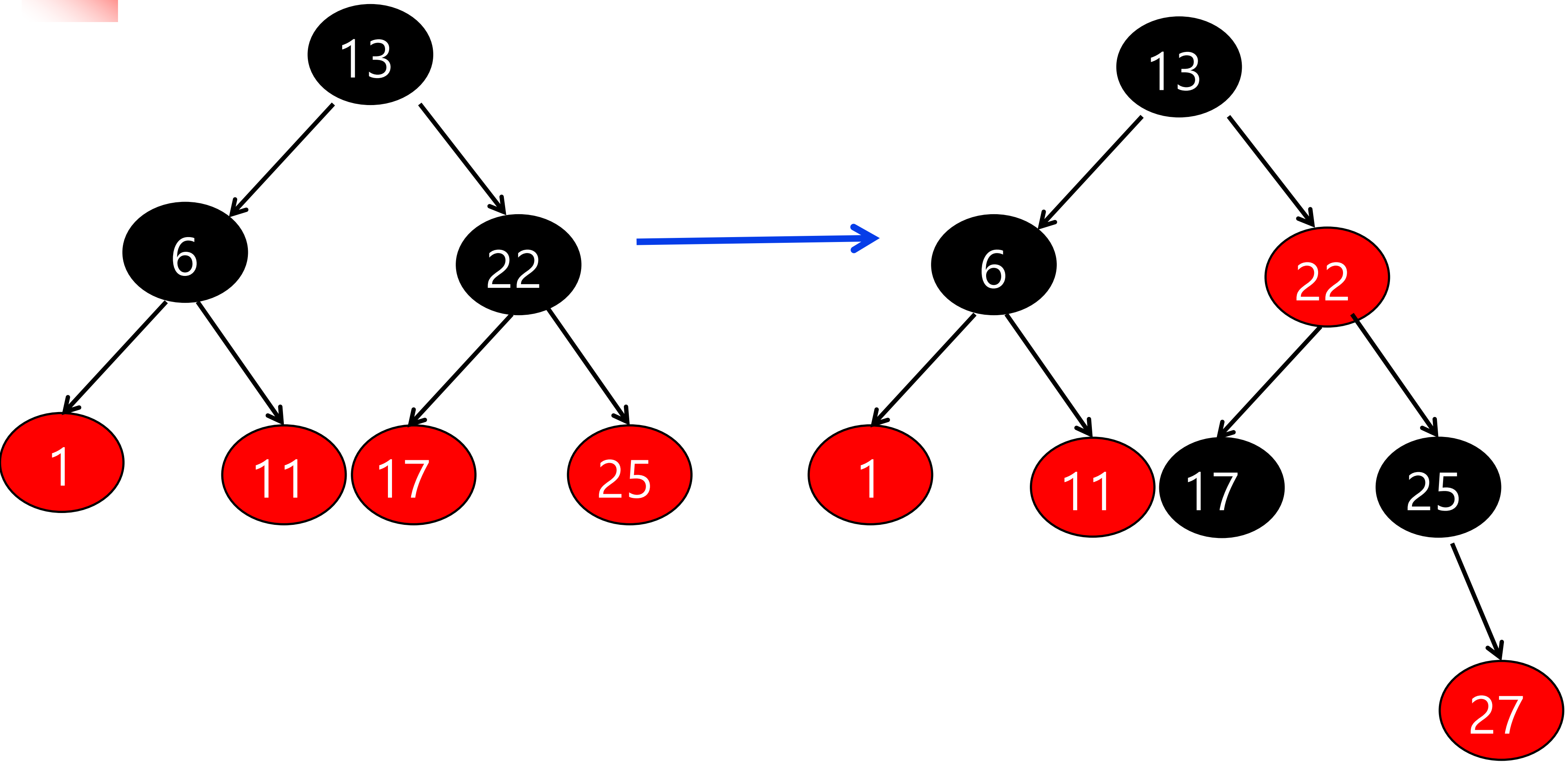
Solution: Insert 6 (2 rotations) (case 5: mirror case)



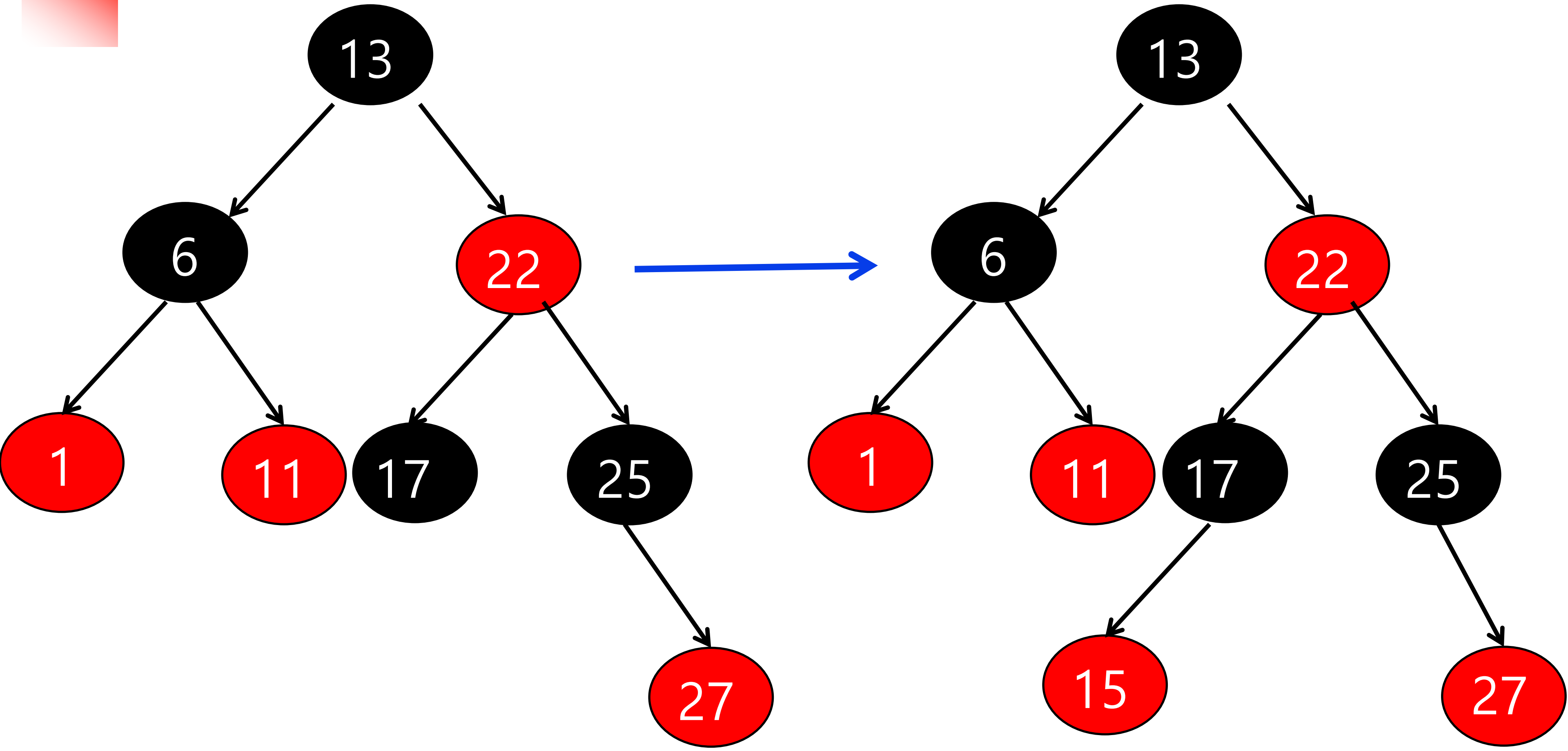
Solution: Insert 27 (case 3)



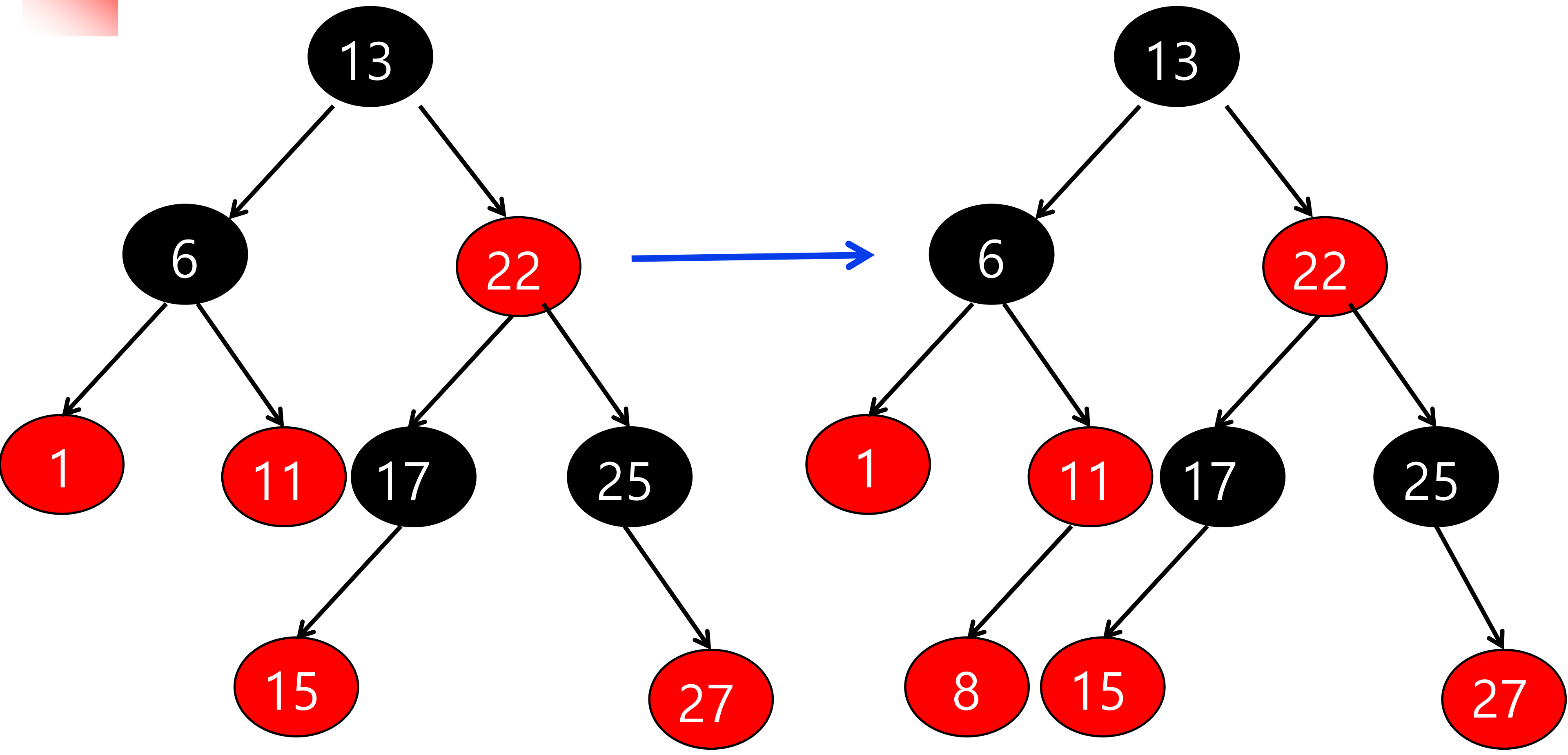
Solution: Insert 27 (case 3)



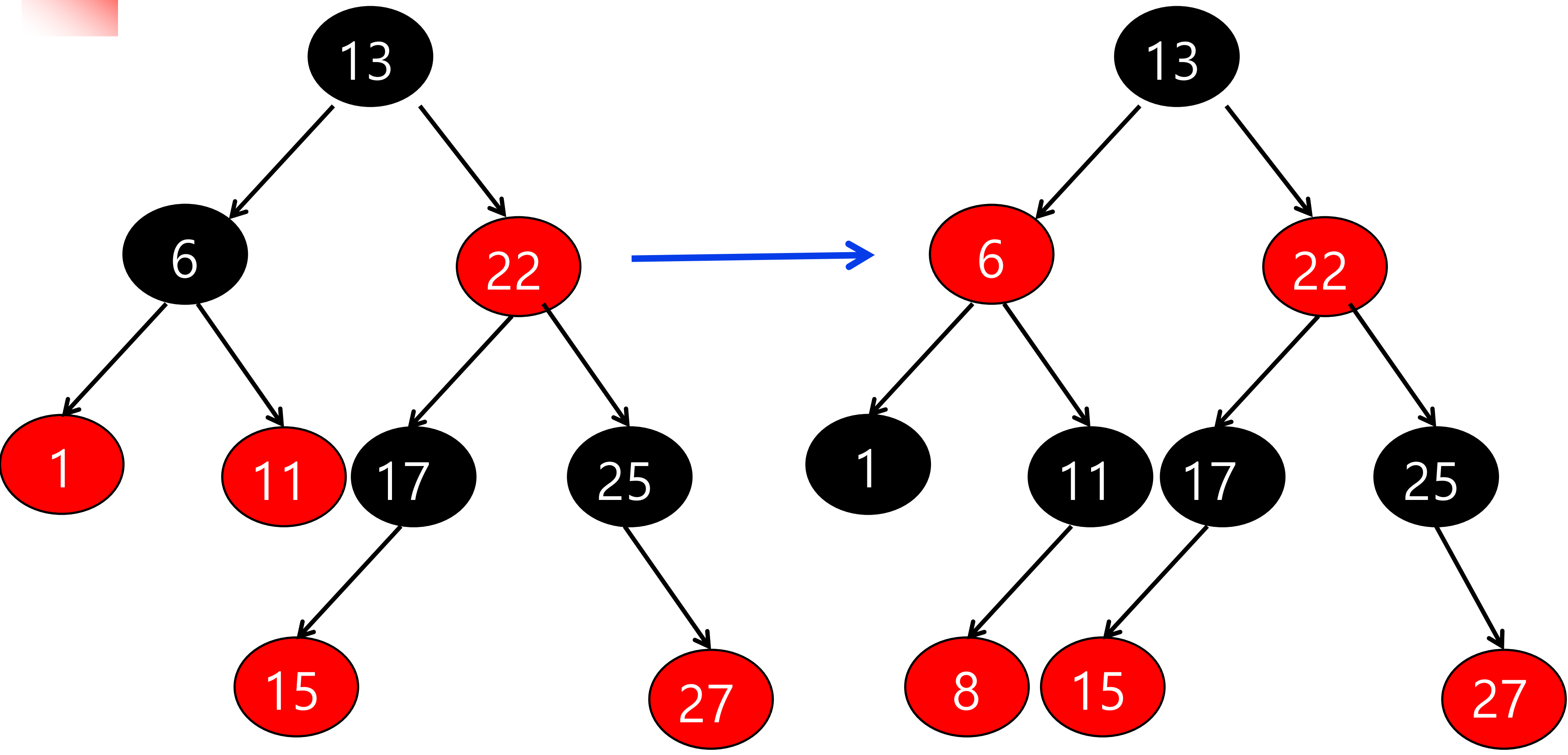
Solution: Insert 15



Solution: Insert 8 (case 3)



Solution: Insert 8 (case 3)



Deletion

- Delete a node as with a regular binary search tree.
- The result is always the deletion of a node that is **a leaf** or has **only one child**.
- To rebalance the tree after deletion, check the color of the sibling to decide the appropriate case.

Deletion

- 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

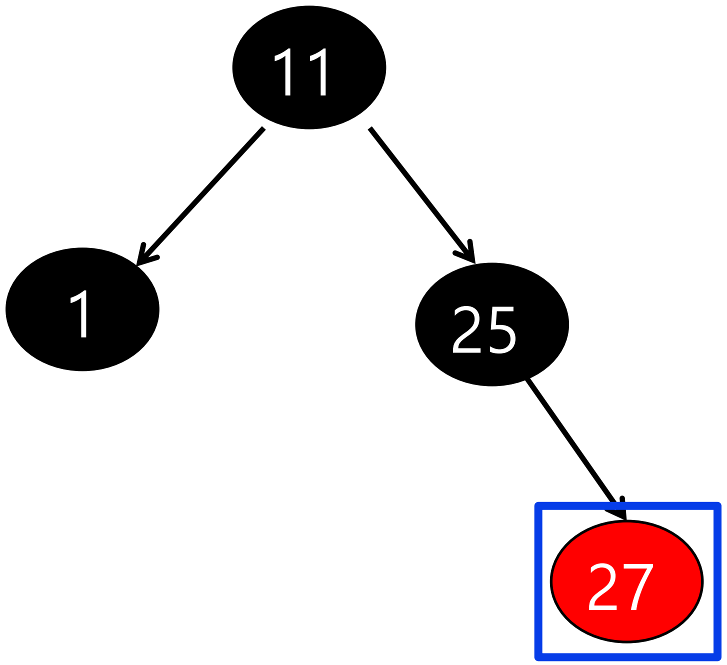
- **D**: a node to be deleted by BST deletion
- **C**: a selected child of **D**

Case 1

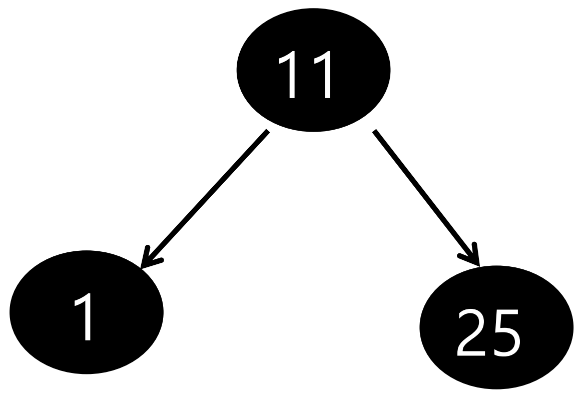
- **D** is a red.
  - **D** must be a leaf, because
    - **D** must be a leaf or has only one child.
    - A red node must have two black children.

  - Simply delete **D**.

Example: Delete 27



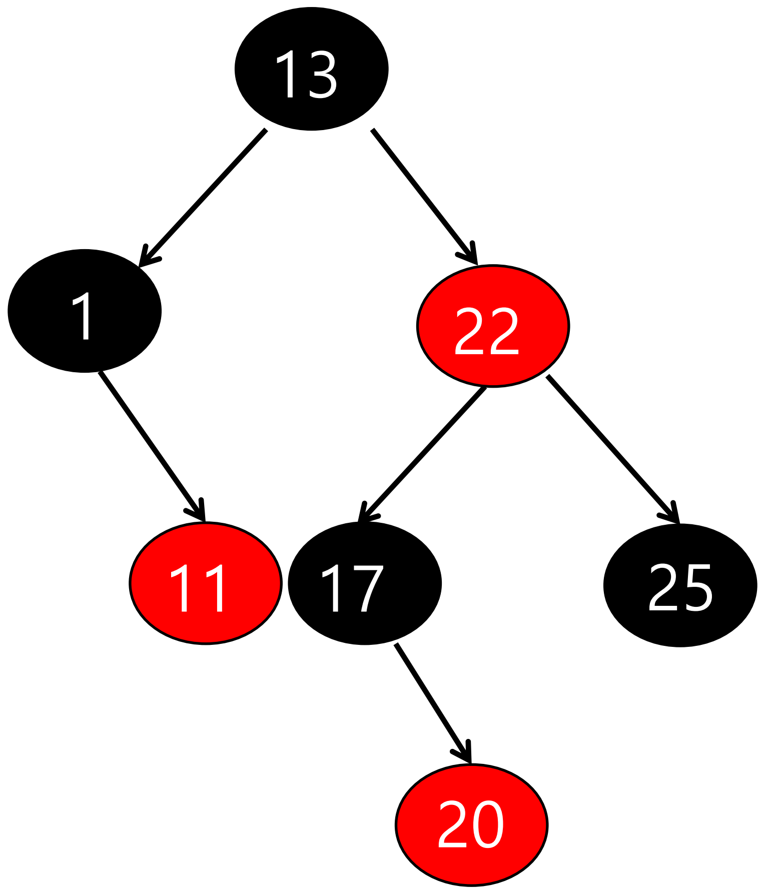
Example: Delete 27



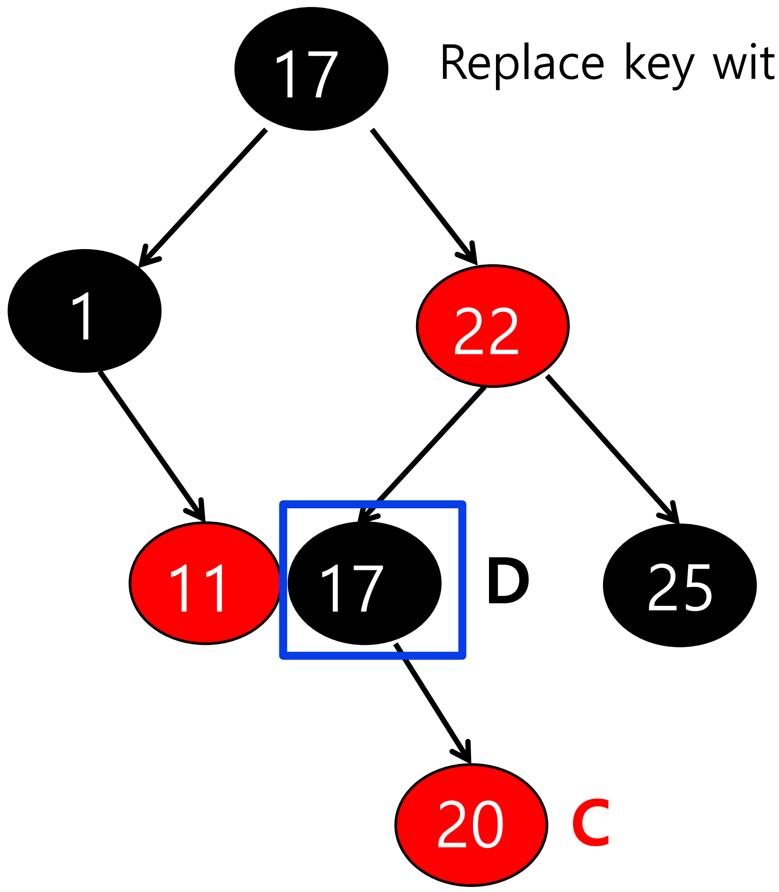
Case 2

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

Example: Delete 13

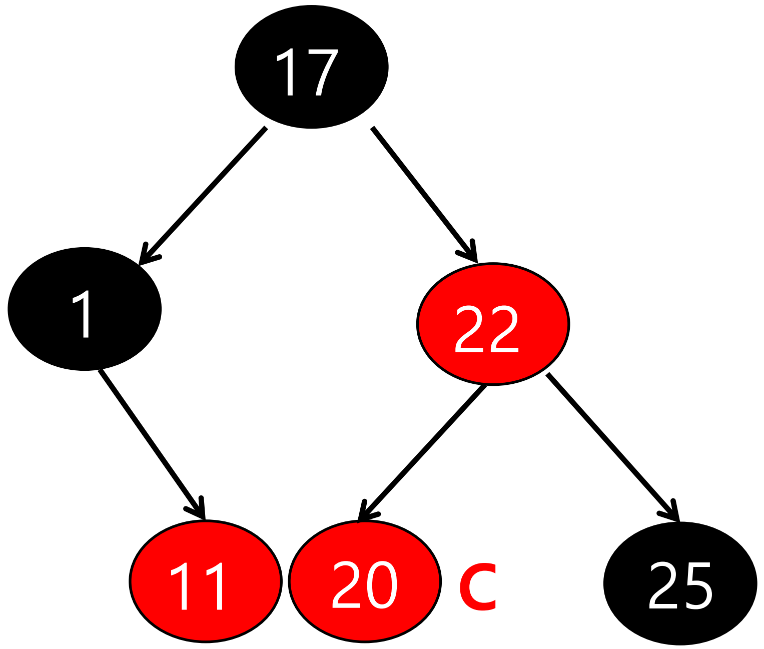


Example: Delete 13



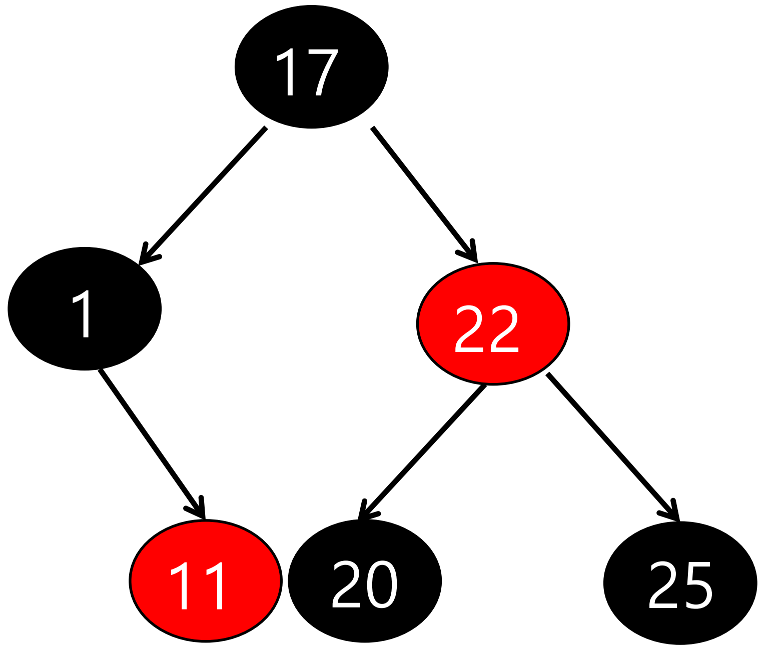
Replace key with 17 (inorder successor)

Example: Delete 13



Replace it with C

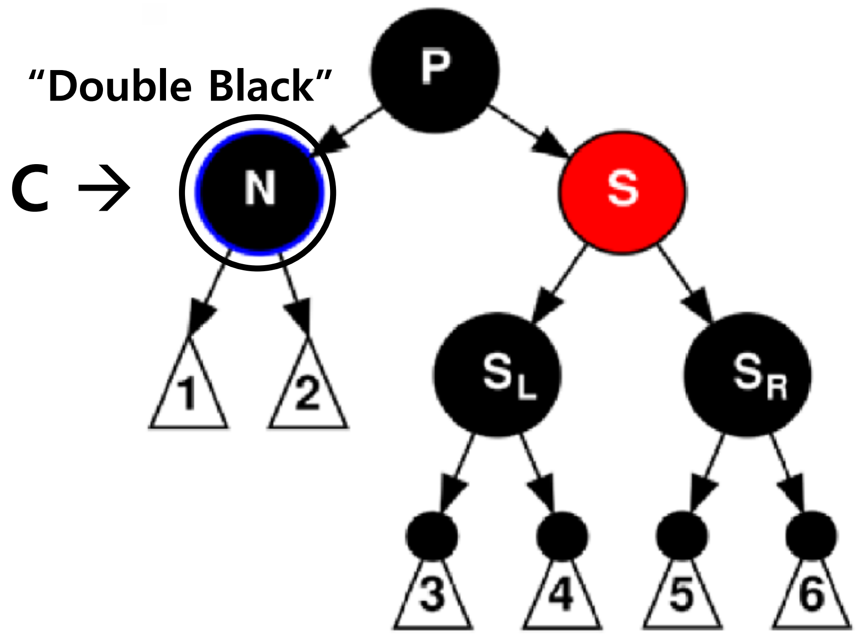
Example: Delete 13



Recolor **C** black

Case 3:

- Both **D** and **C** are black.
- Includes when **D** is a black leaf (NIL is **C**).
- Delete **D**, and replace it with **C**.
- Relabel **C** as **N**.
- **N is double black.**



Case 3 (contd.)

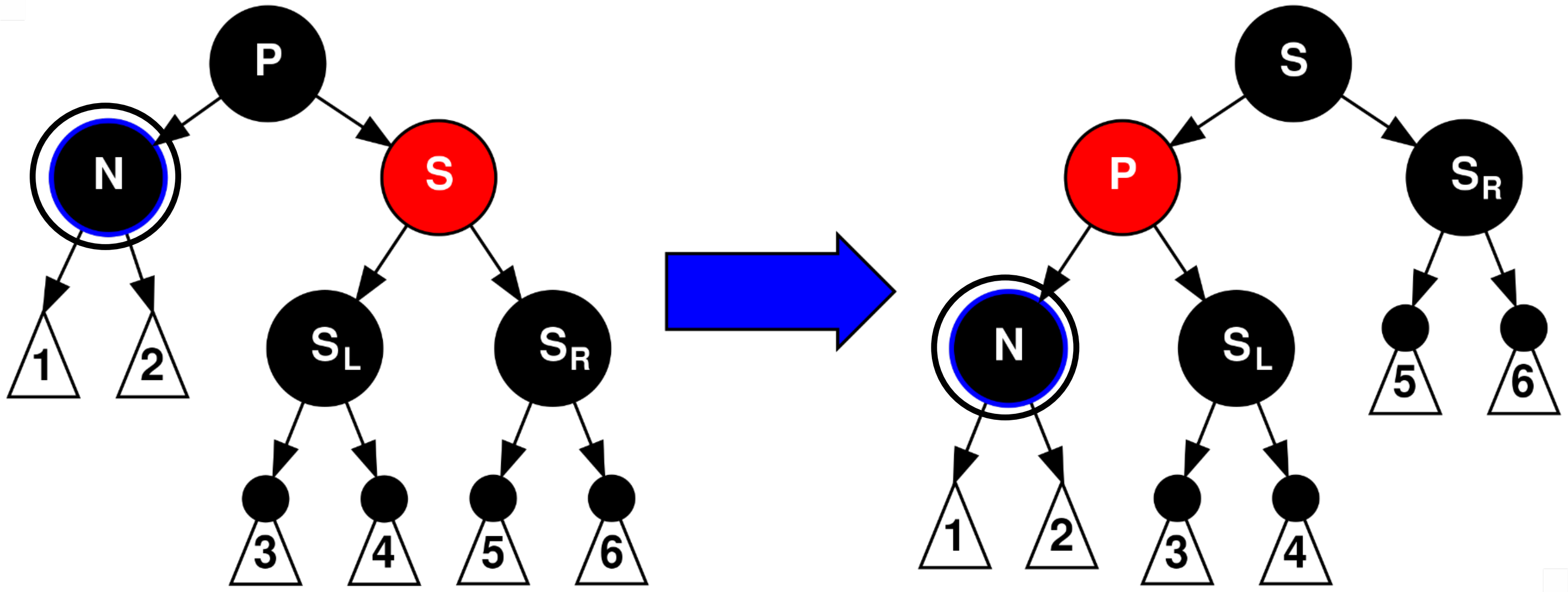
- Double black needs to be removed to single black to restore balance.
- 6 different cases to consider
  - Some are terminal cases.
  - Others are non-terminal cases that need further operations.

Case 3-1

- **N** is a new root.
- Remove double black to single black.
- Terminal state (Done!)

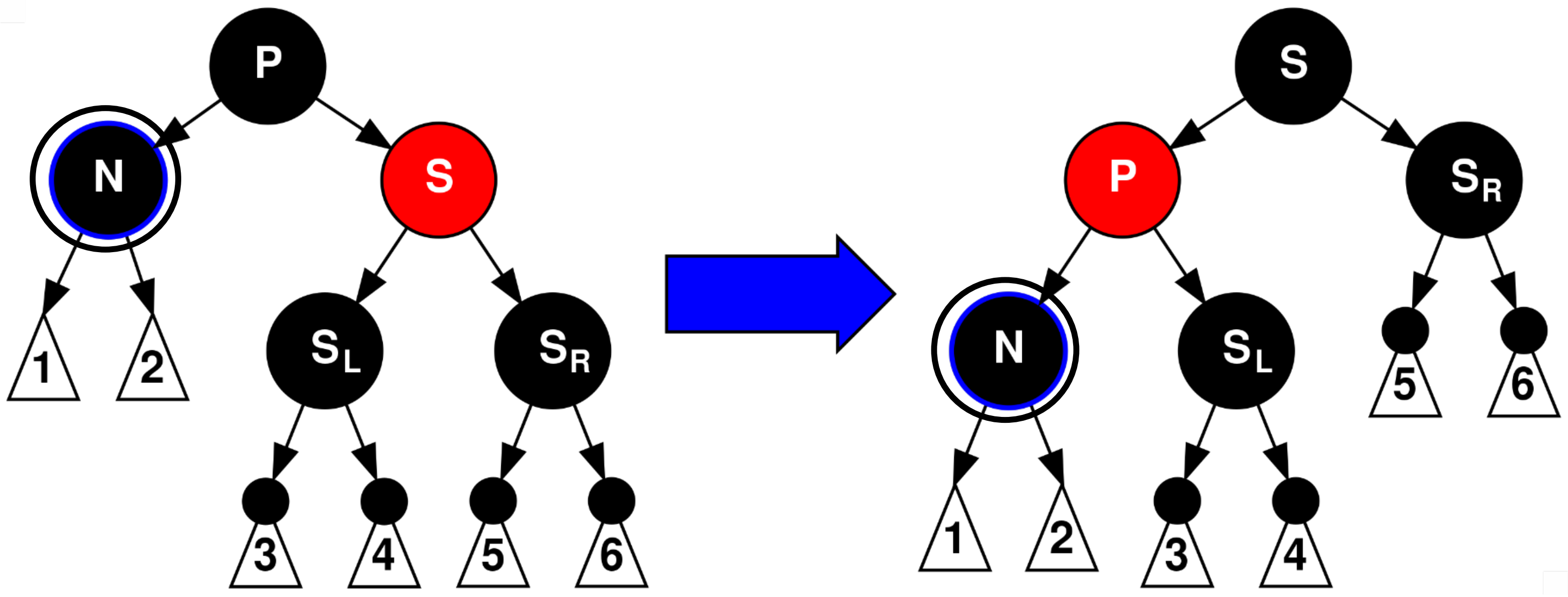
Case 3-2

- Sibling **S** is red.
- Parent **P** and both of **S**'s children **S_L** and **S_R** must be black to be a RB-tree.
- Switch the colors of **P** and **S**, and then *rotate toward* **N** at **P**.

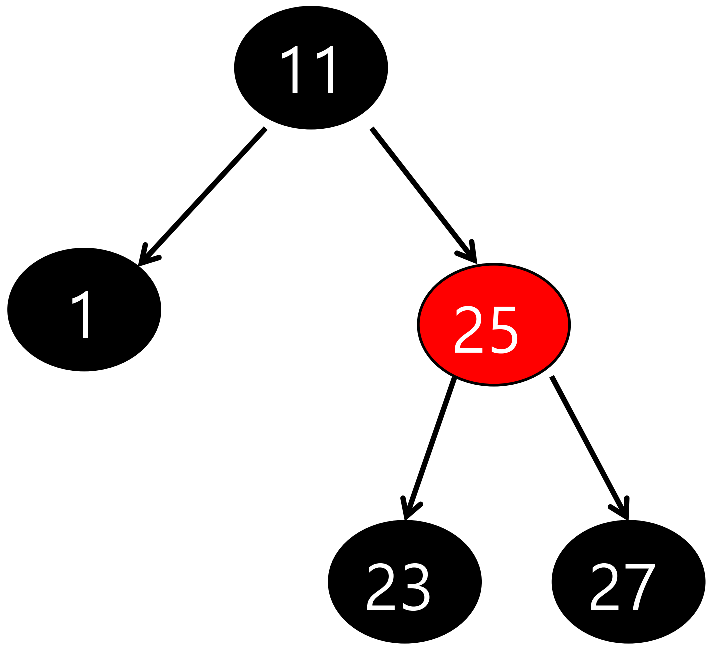


Case 3-2 (contd.)

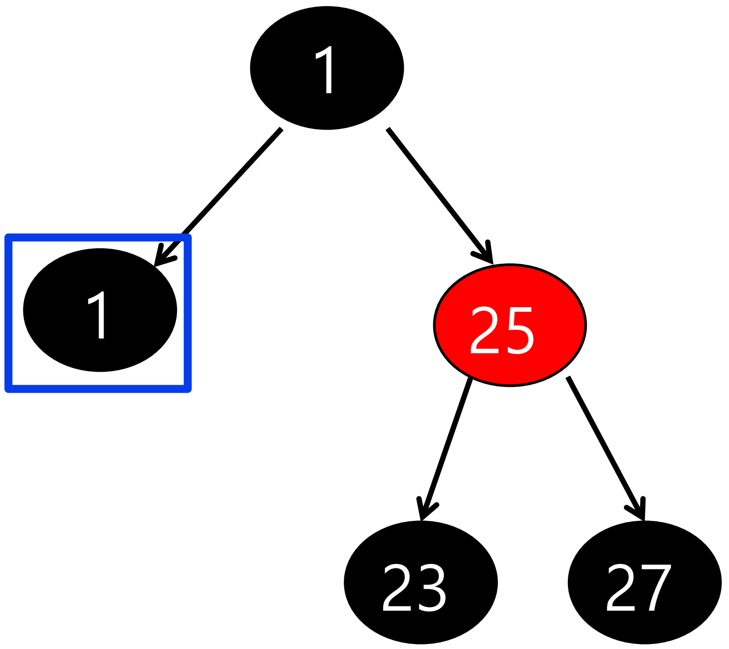
- Non-terminal state
- **N** is still double black.
- Non-terminal state: Proceed to a different case.



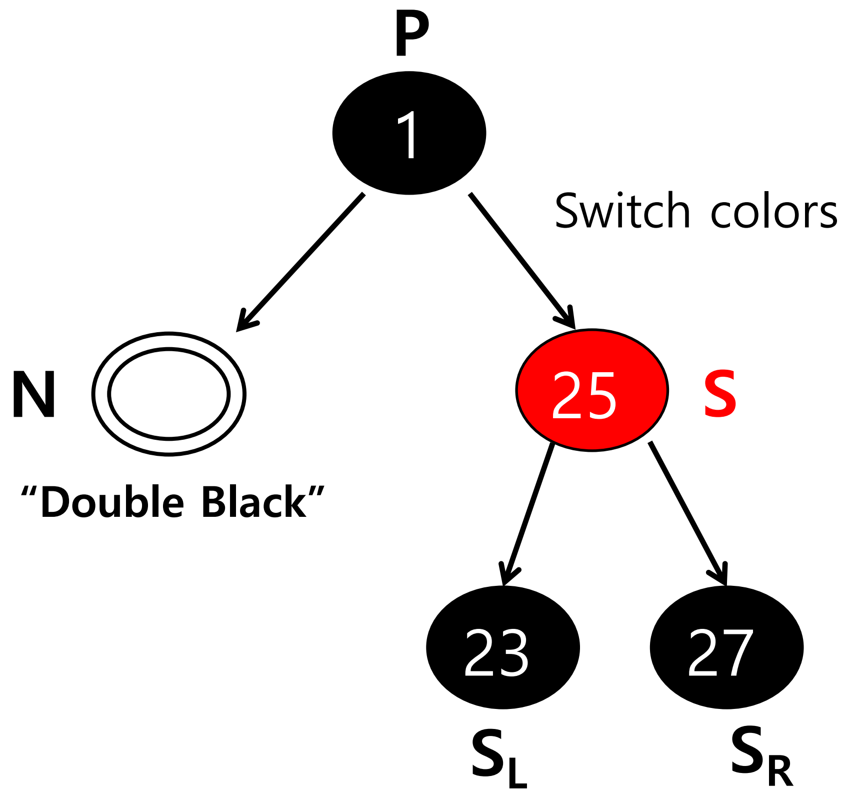
Example: Delete 11



Example: Delete 11

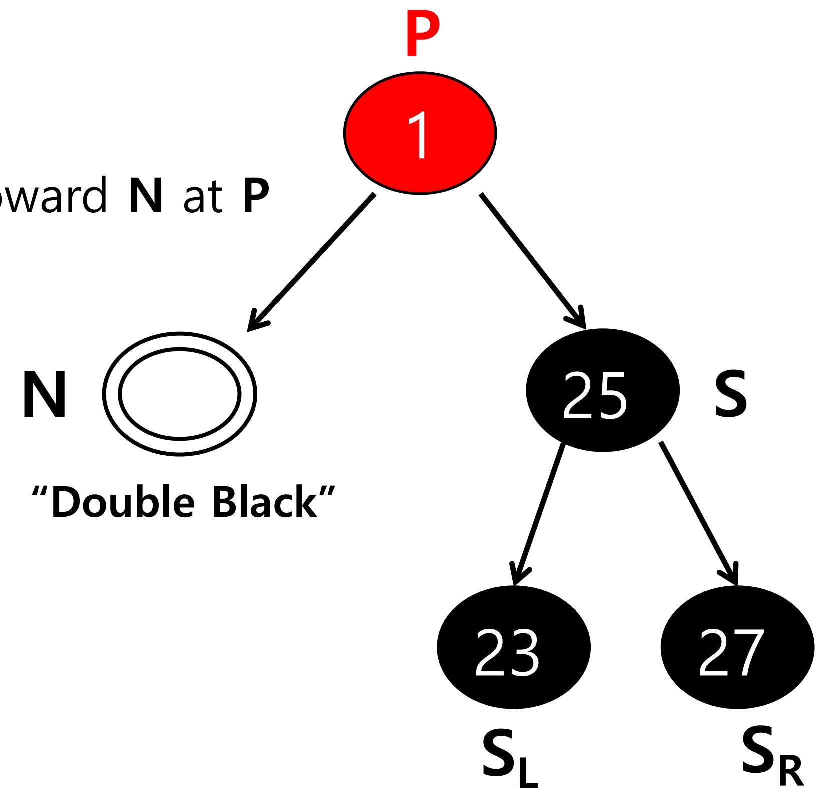


Example: Delete 11

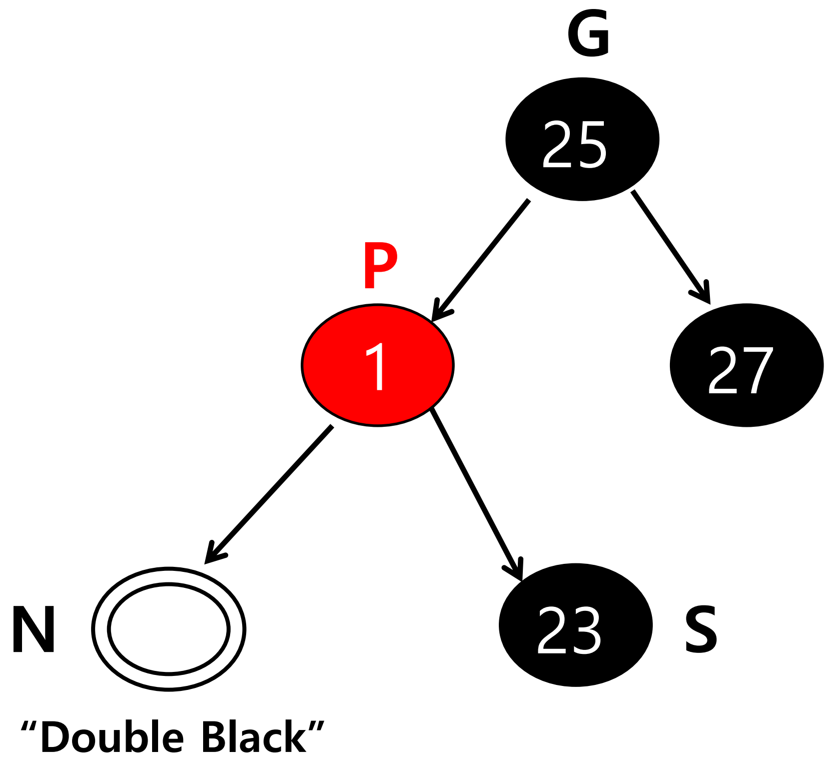


Example: Delete 11

Rotated toward **N** at **P**

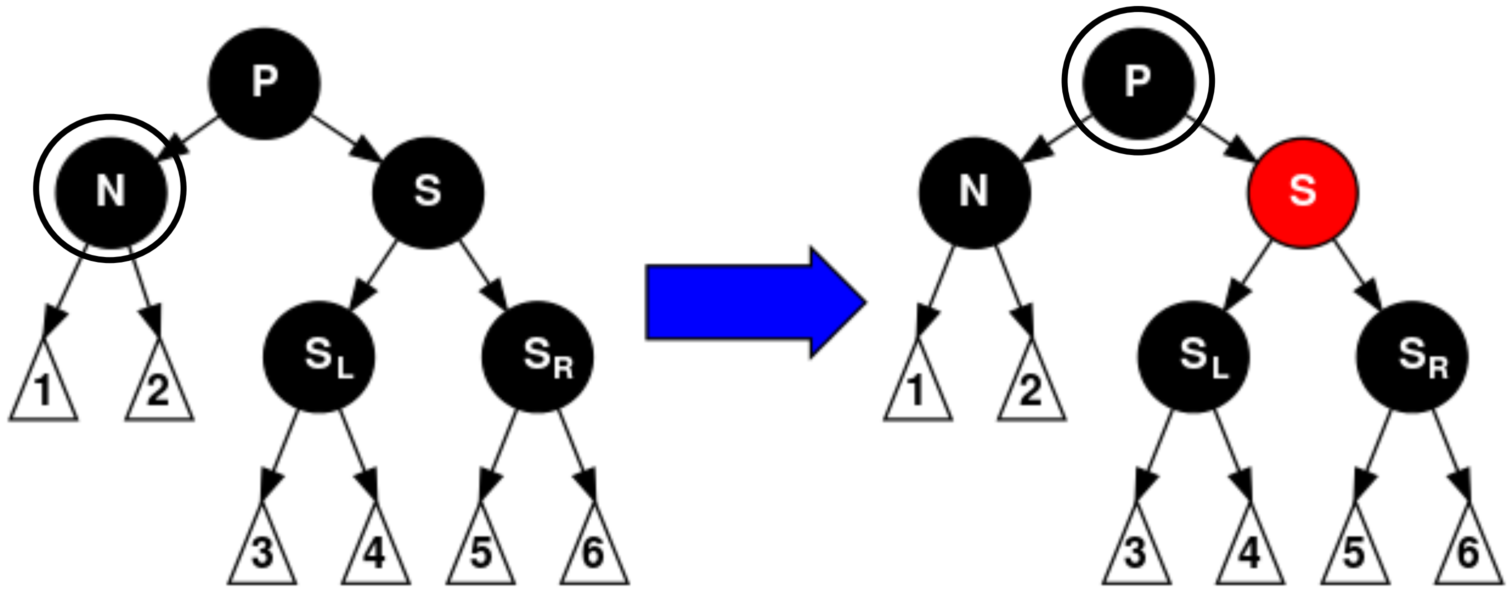


Example: Delete 11

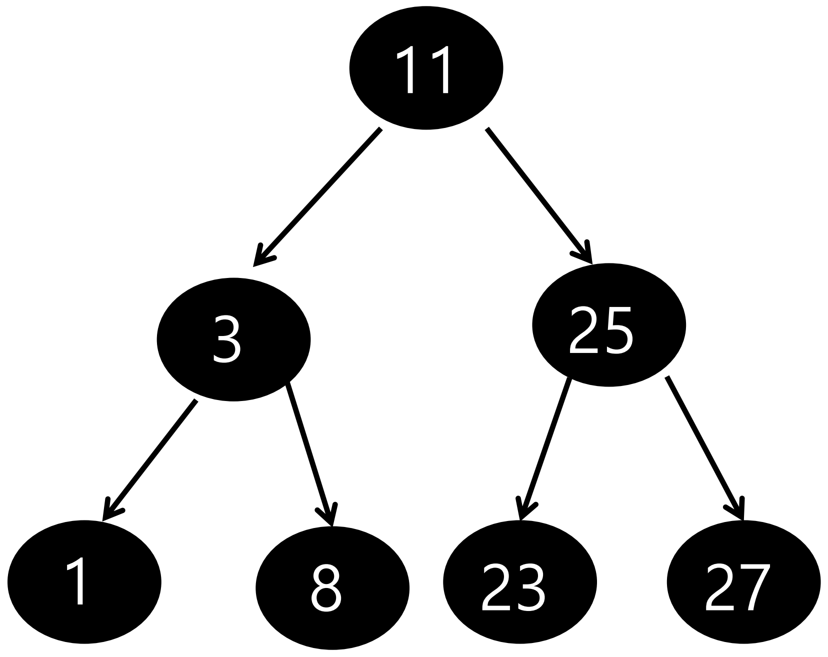


Case 3-3

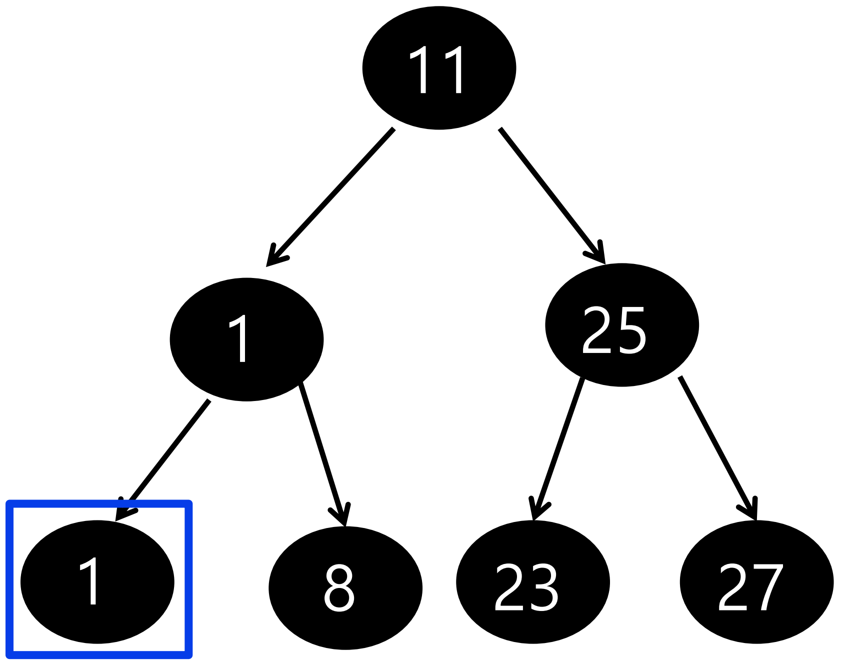
- Parent **P**, Sibling **S**, **S**'s children **S_L** and **S_R** are all black.
- Recolor **S** red.
- Move double black to **P**.
- Non-terminal state.



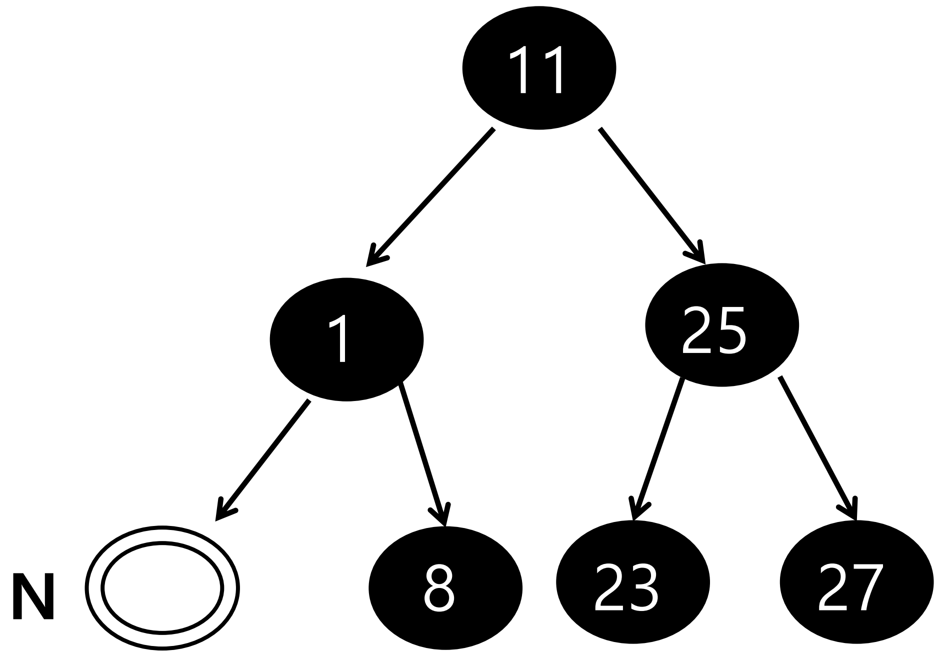
Example: Delete 3



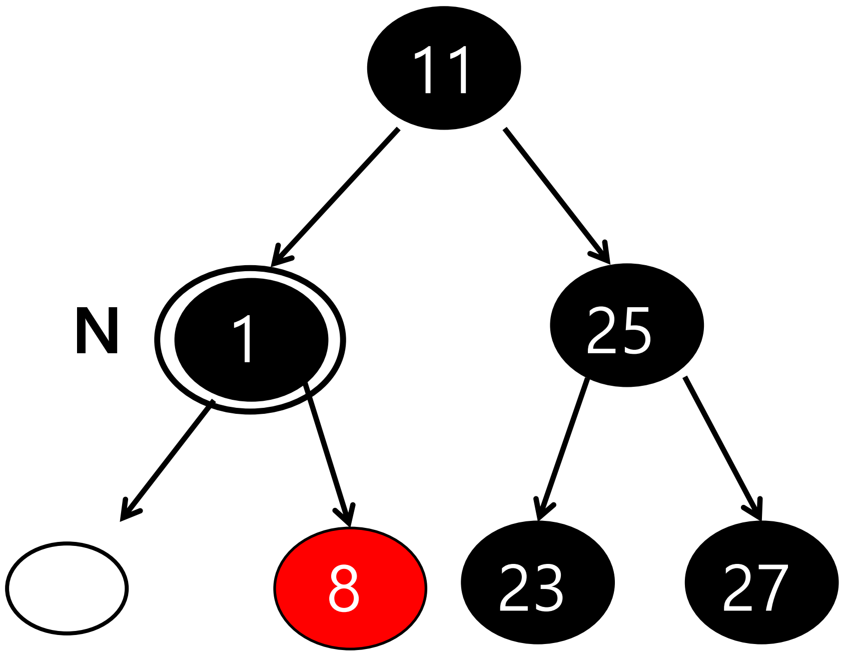
Example: Delete 3



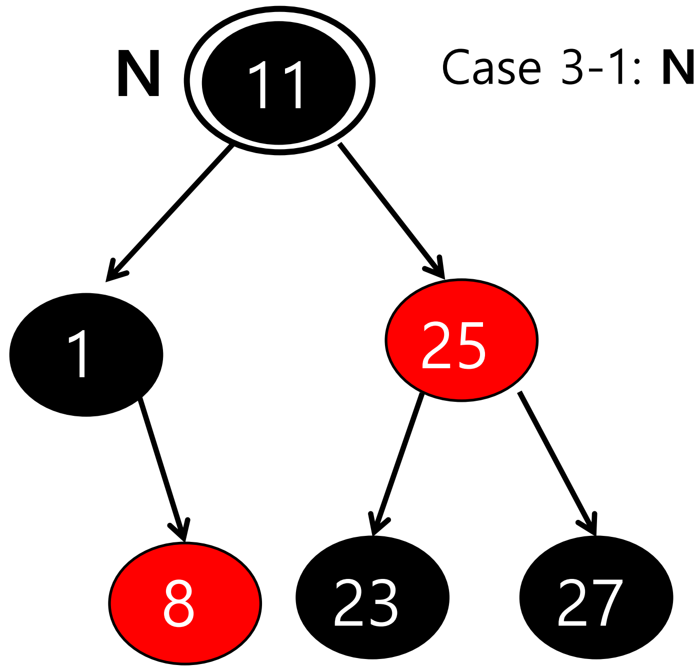
Example: Delete 3



Example: Delete 3



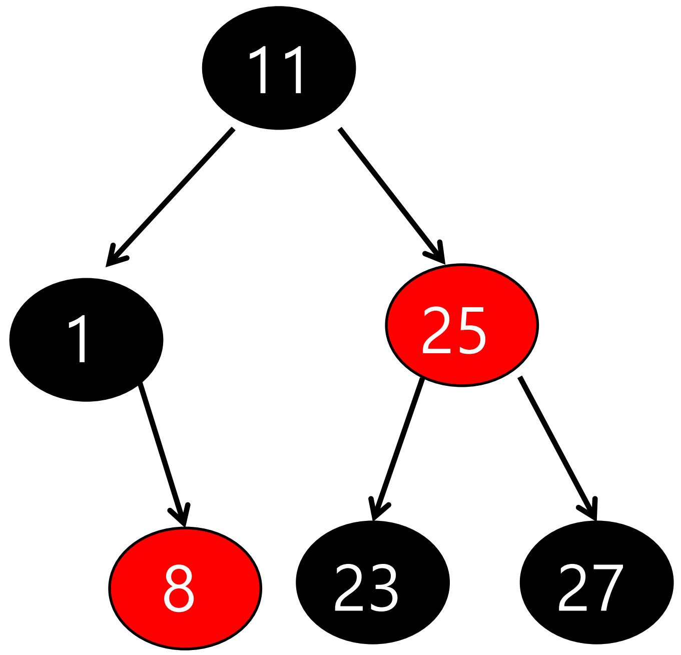
Example: Delete 3



Case 3-1: **N** is a new root.

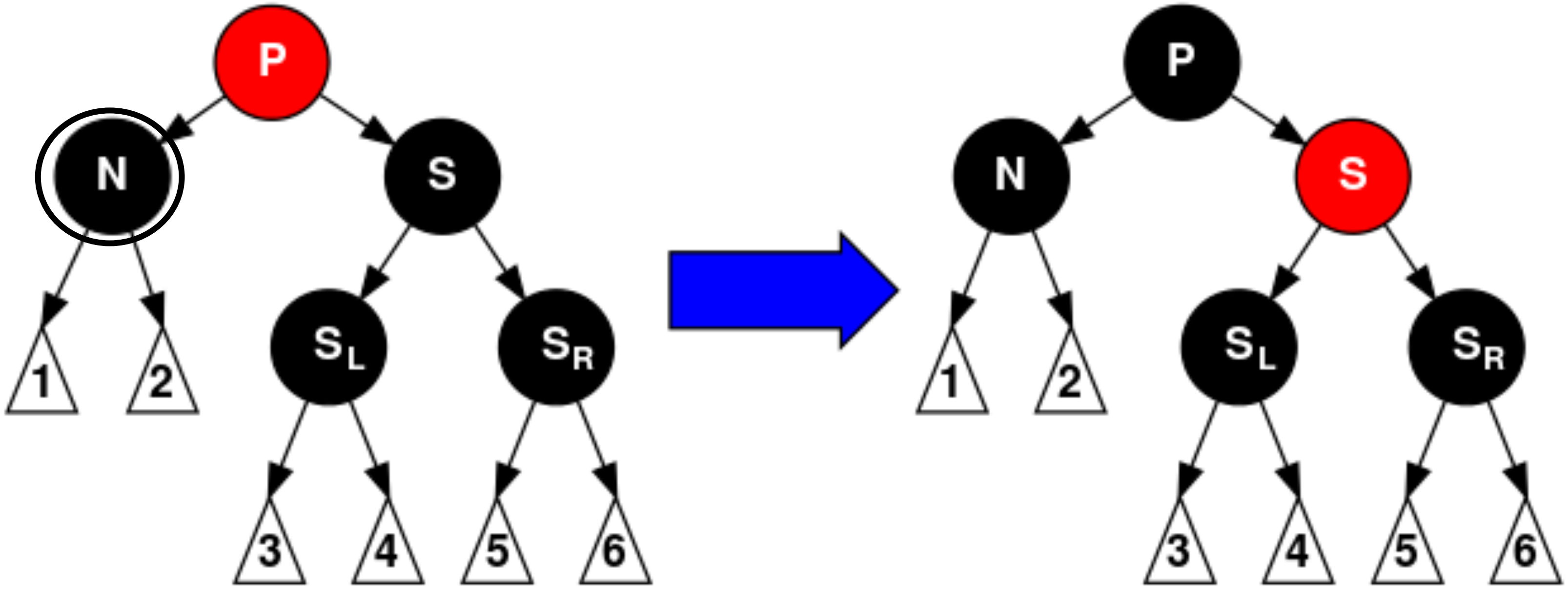
Example: Delete 3

A terminal state

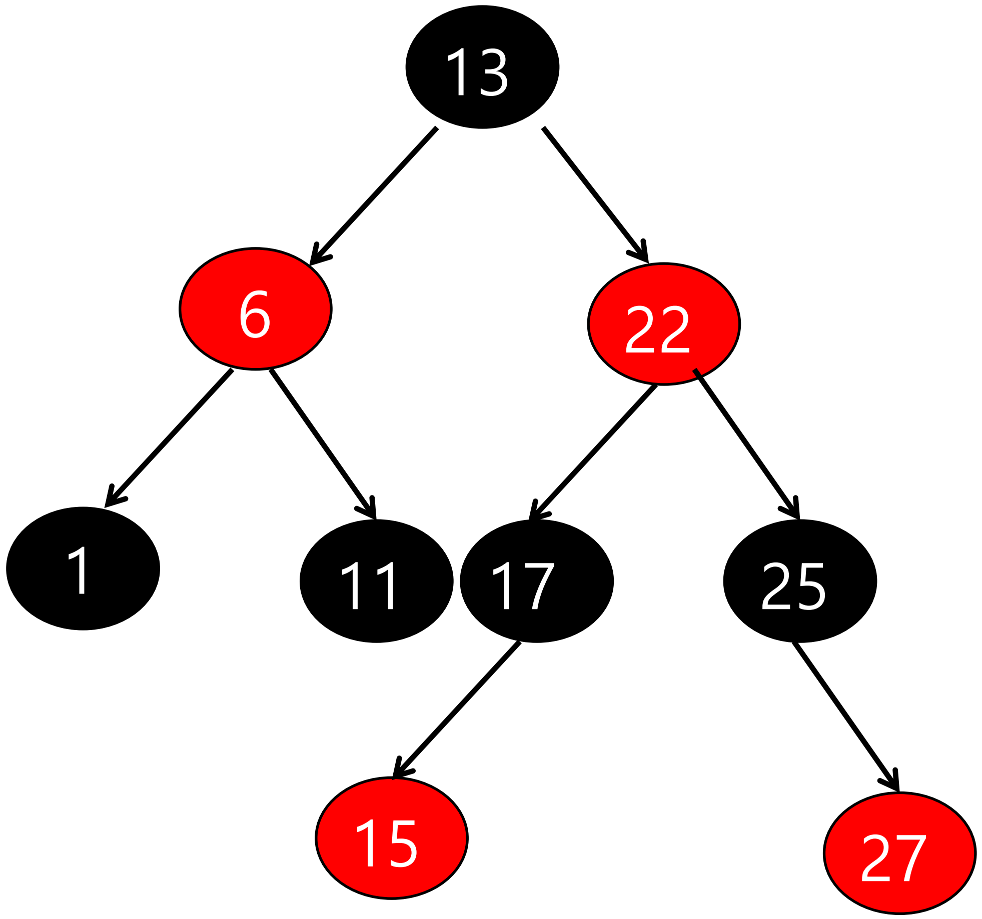


Case 3-4

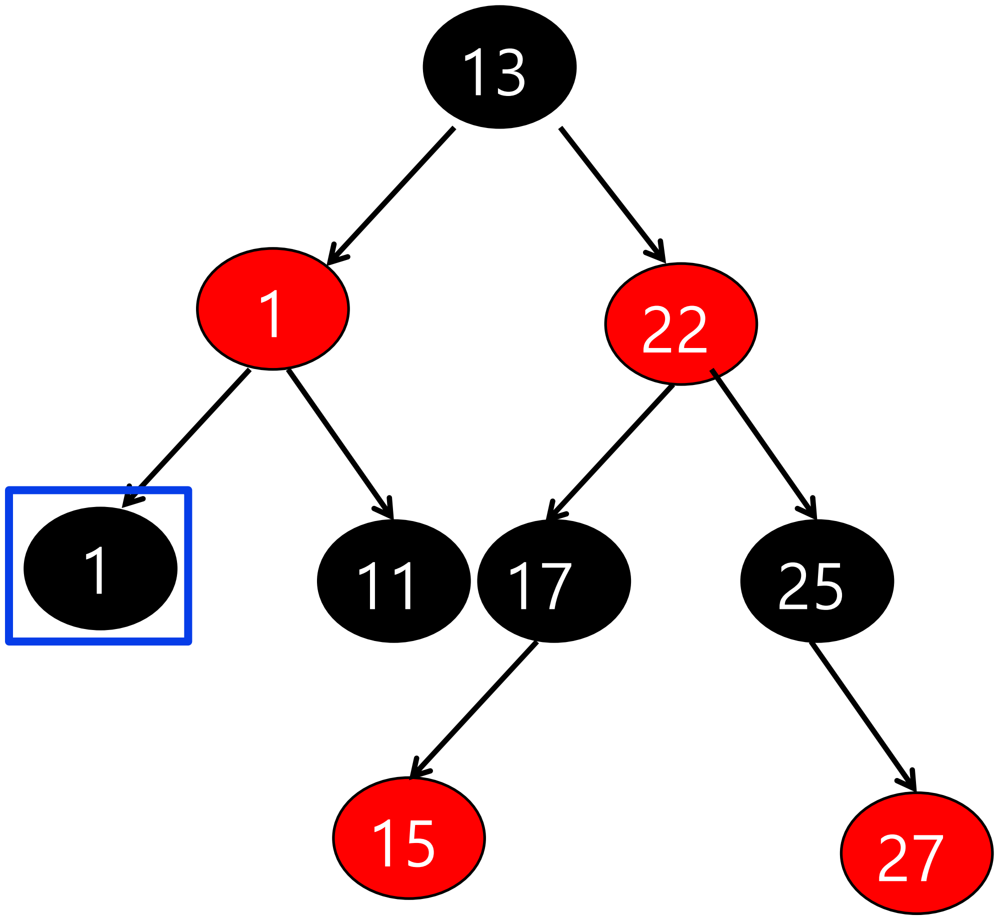
- Parent **P** is red, but Sibling **S**, **S**'s children **S_L** and **S_R** are black.
- Simply exchange the colors of **S** and **P**.
- Terminal state.



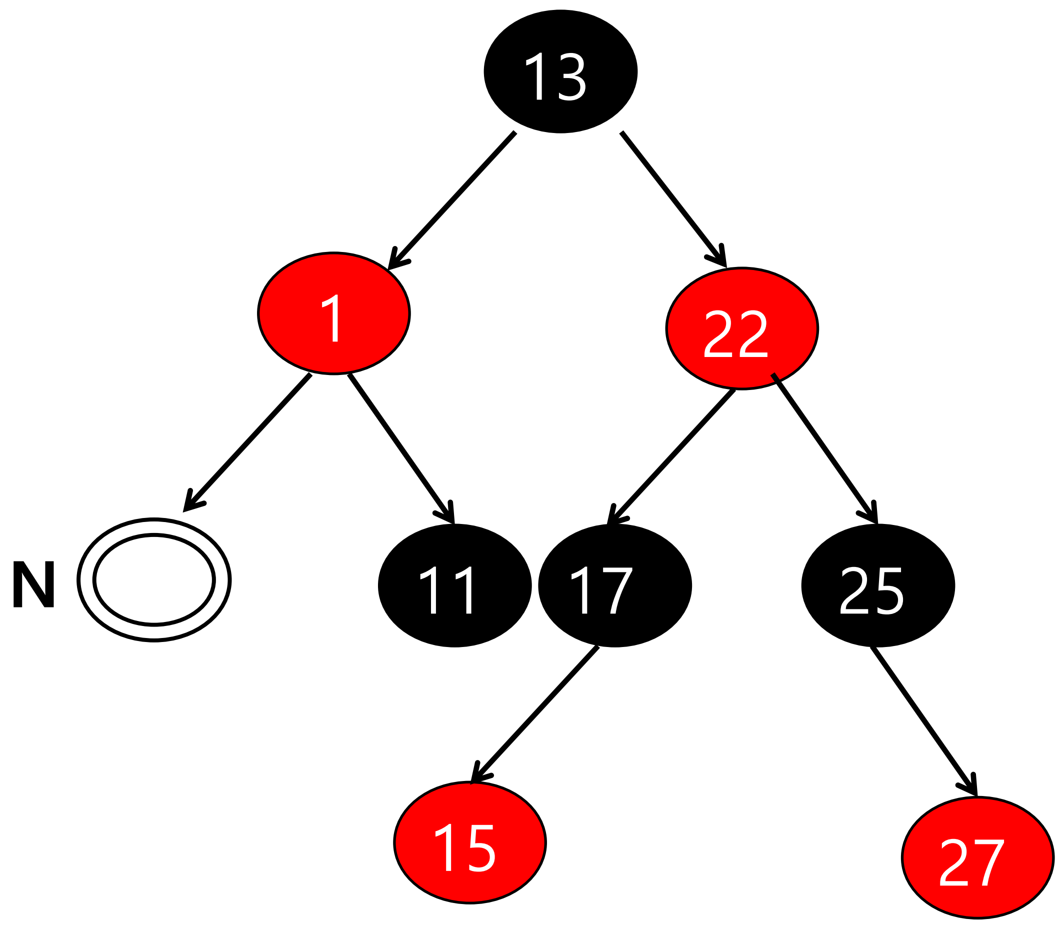
Example: Delete 6



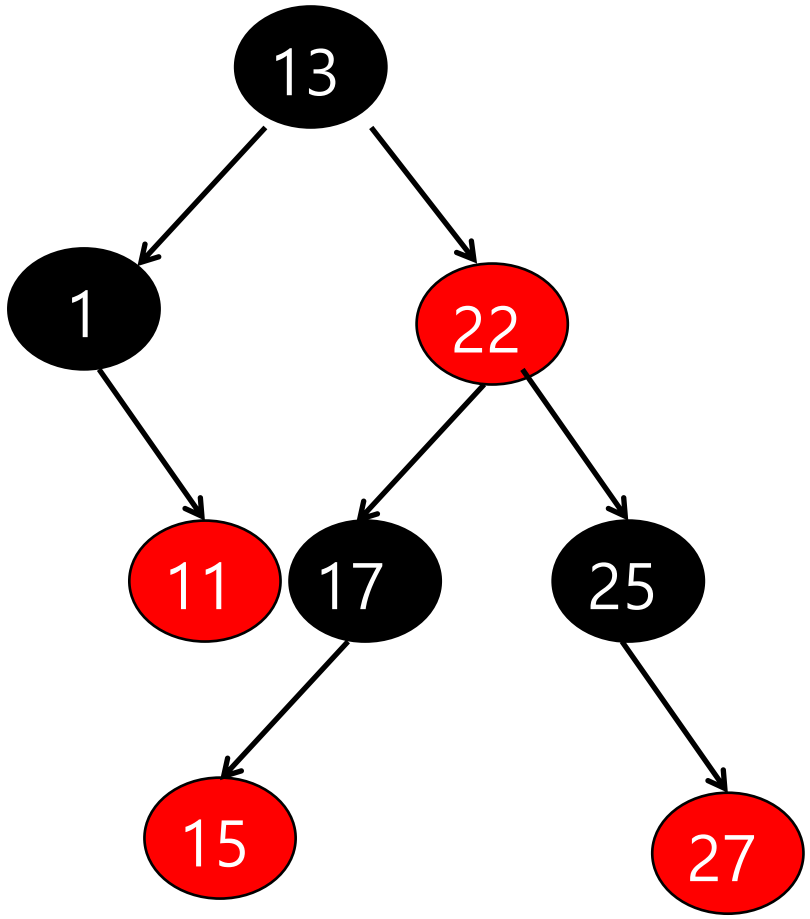
Example: Delete 6



Example: Delete 6

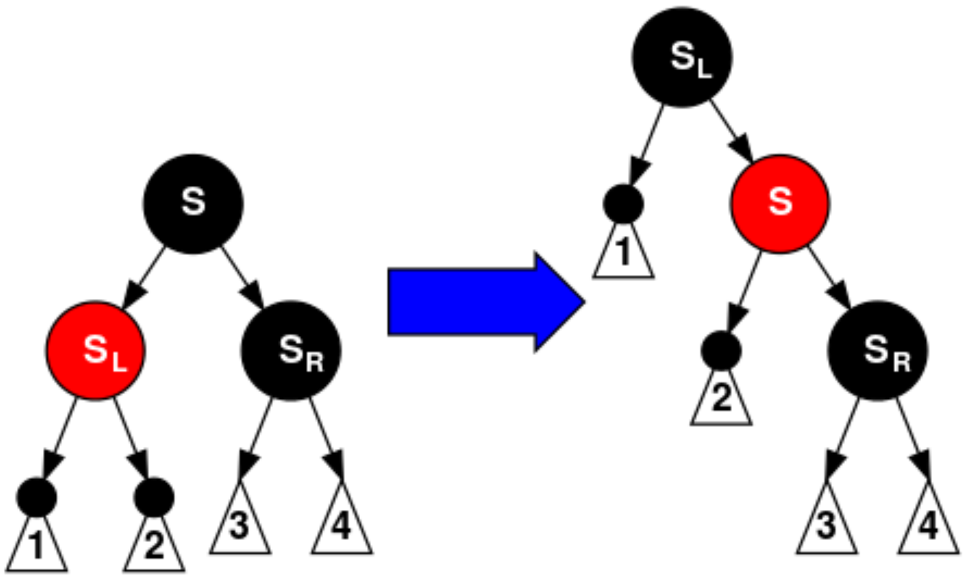


Example: Delete 6



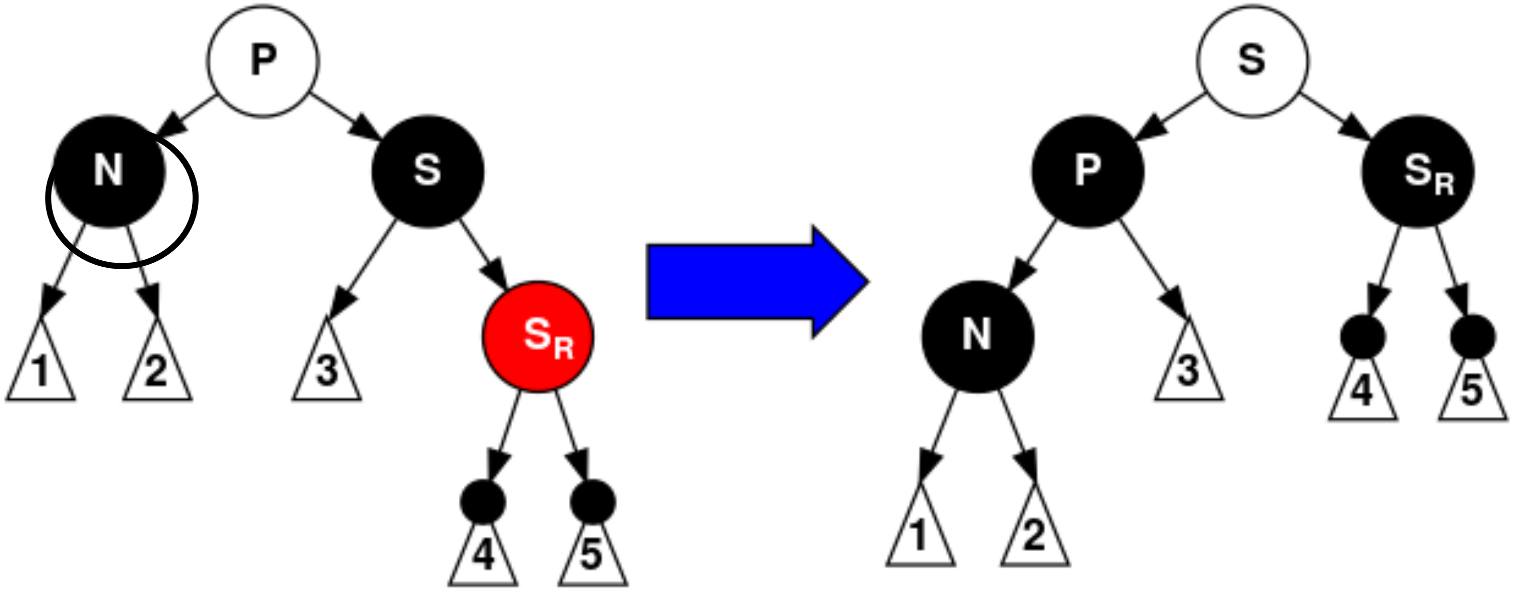
Case 3-5

- 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**.
- Non-terminal state.

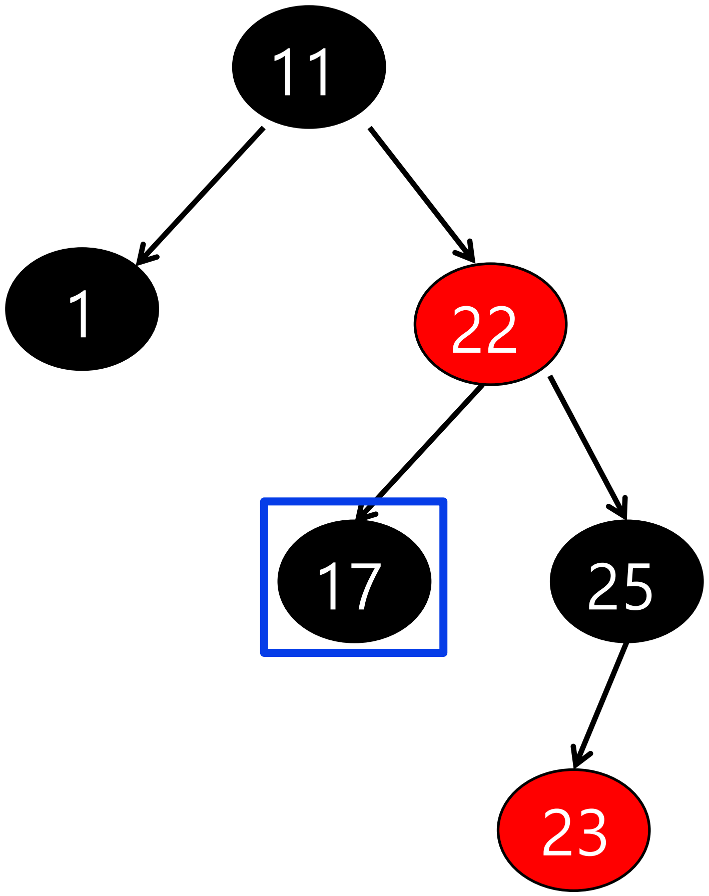


Case 3-6

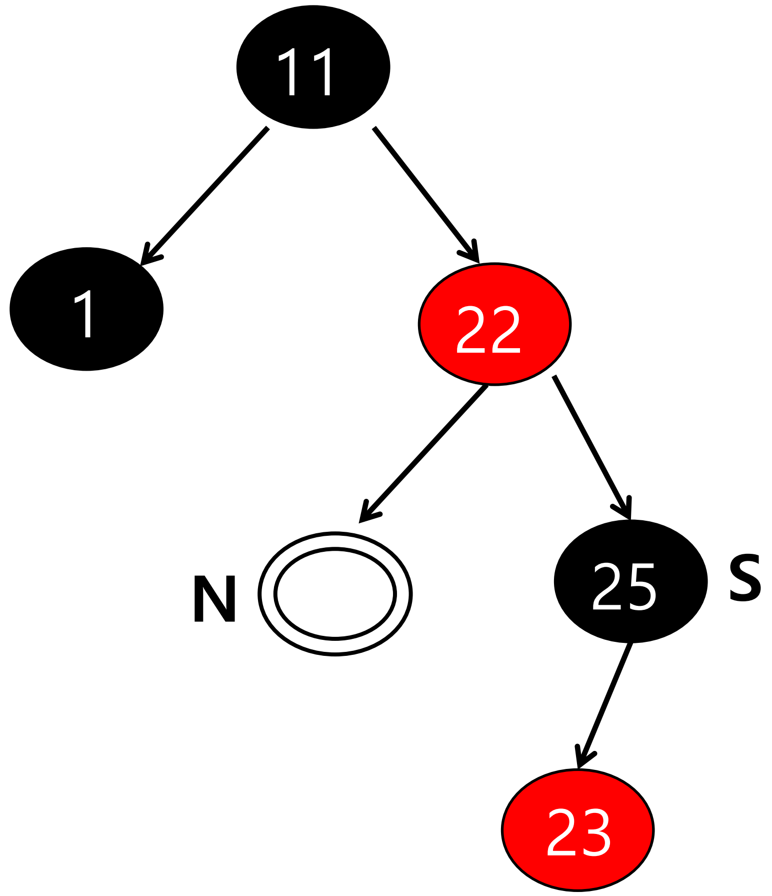
- Sibling **S** is black, **S**’s close child **S_L** is black, and a distant child **S_R** is red.
- Rotate **P** (either color) toward **N**.
- Exchange colors between **P** and **S**.
- Recolor **S_R** black.



Example: Delete 17

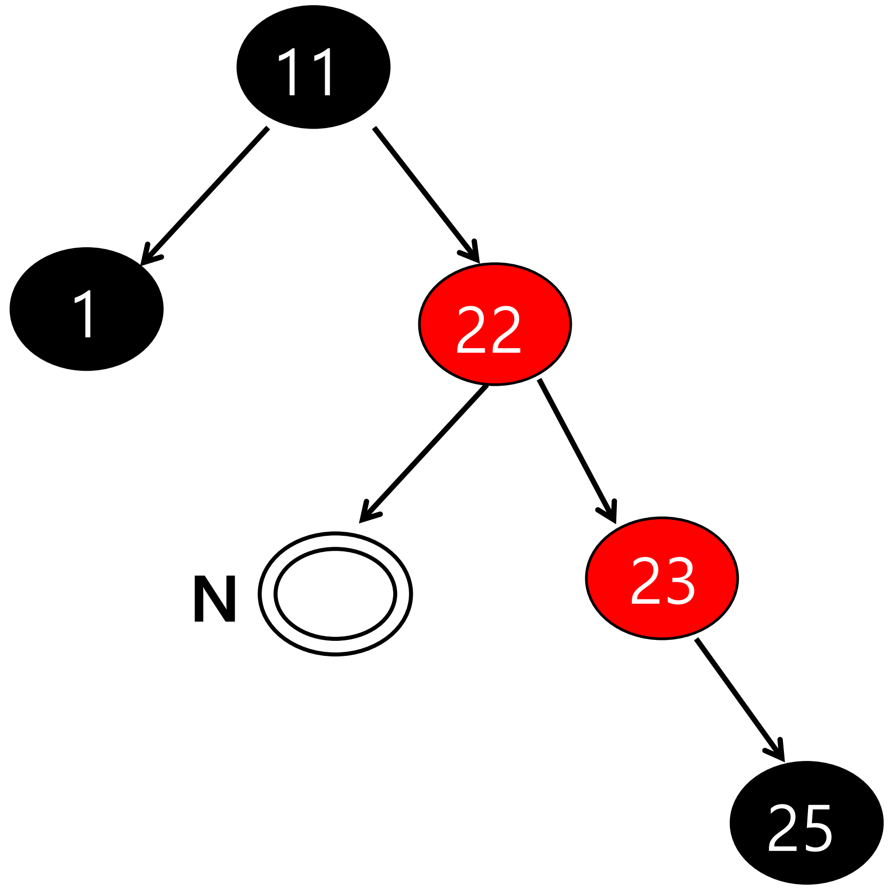


Example: Delete 17



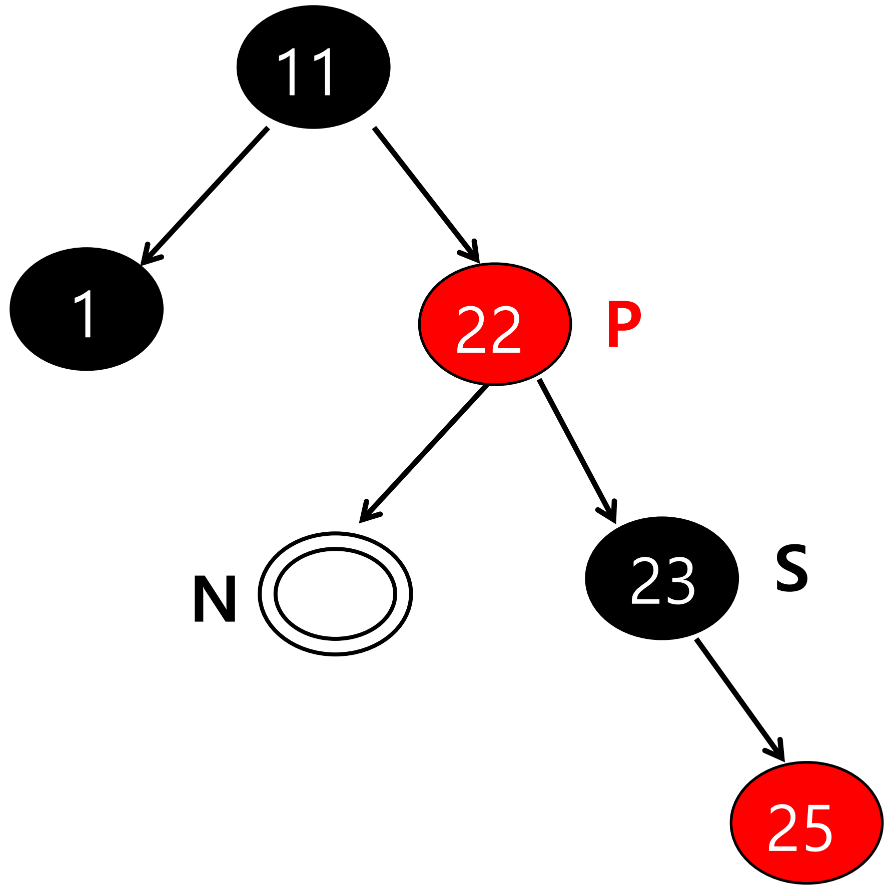
Case 3-5: **S**'s closer child (23) red
Rotate right at 23

Example: Delete 17



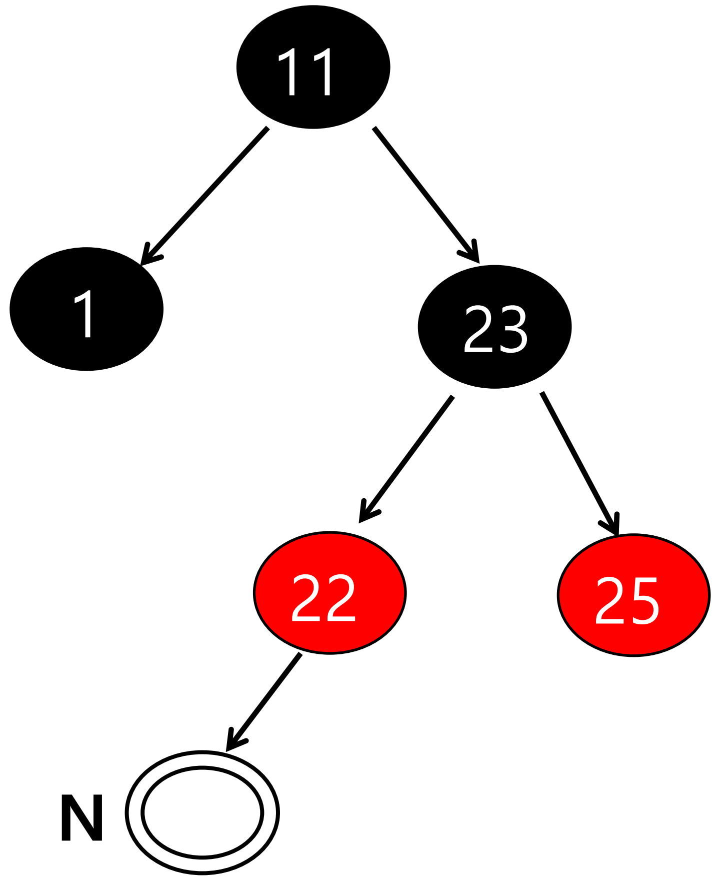
Exchange colors between 23 and 25

Example: Delete 17



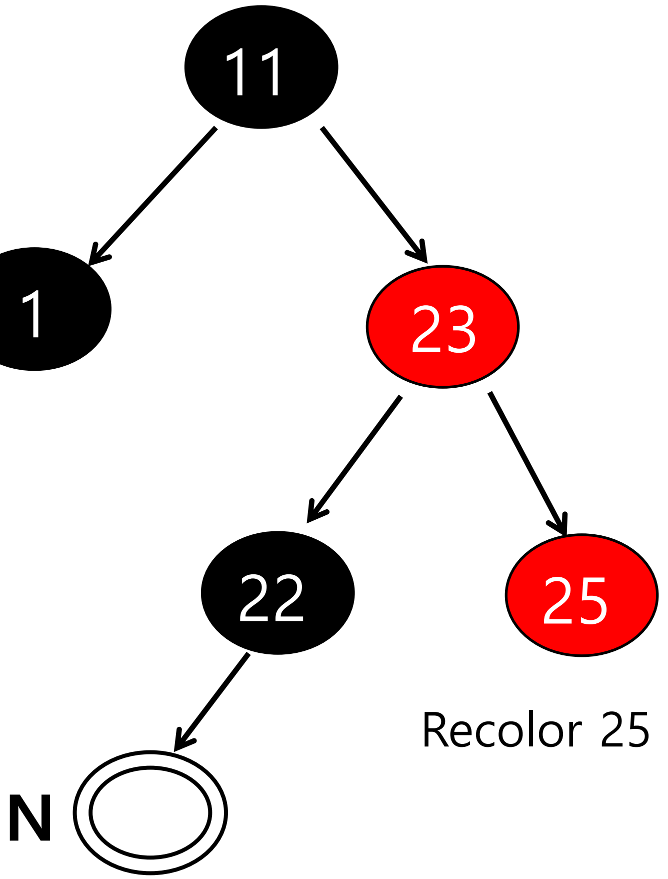
Case 3-6: **S**'s distant child (25) red
Rotate left at **P**

Example: Delete 17



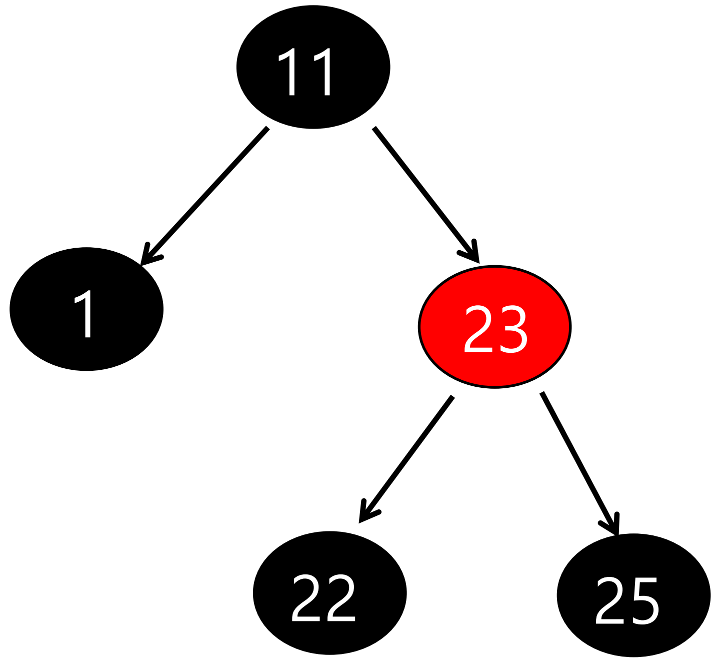
Exchange colors between 22 and 23

Example: Delete 17

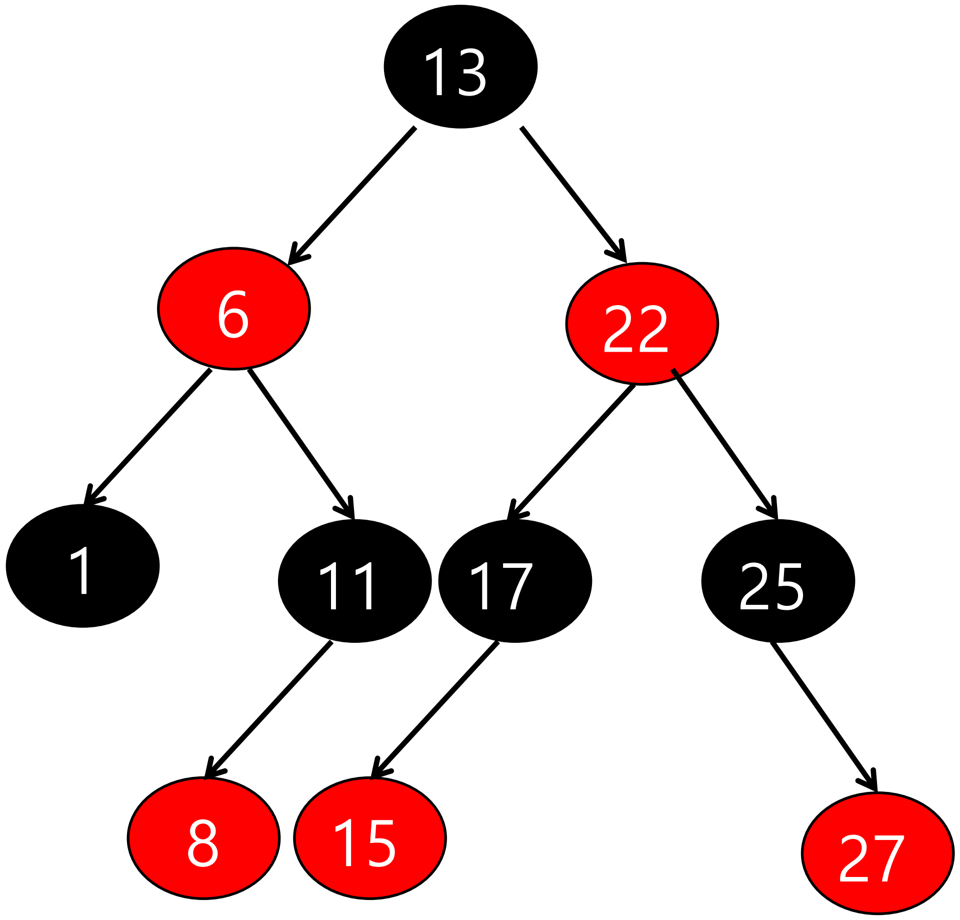


Recolor 25 black

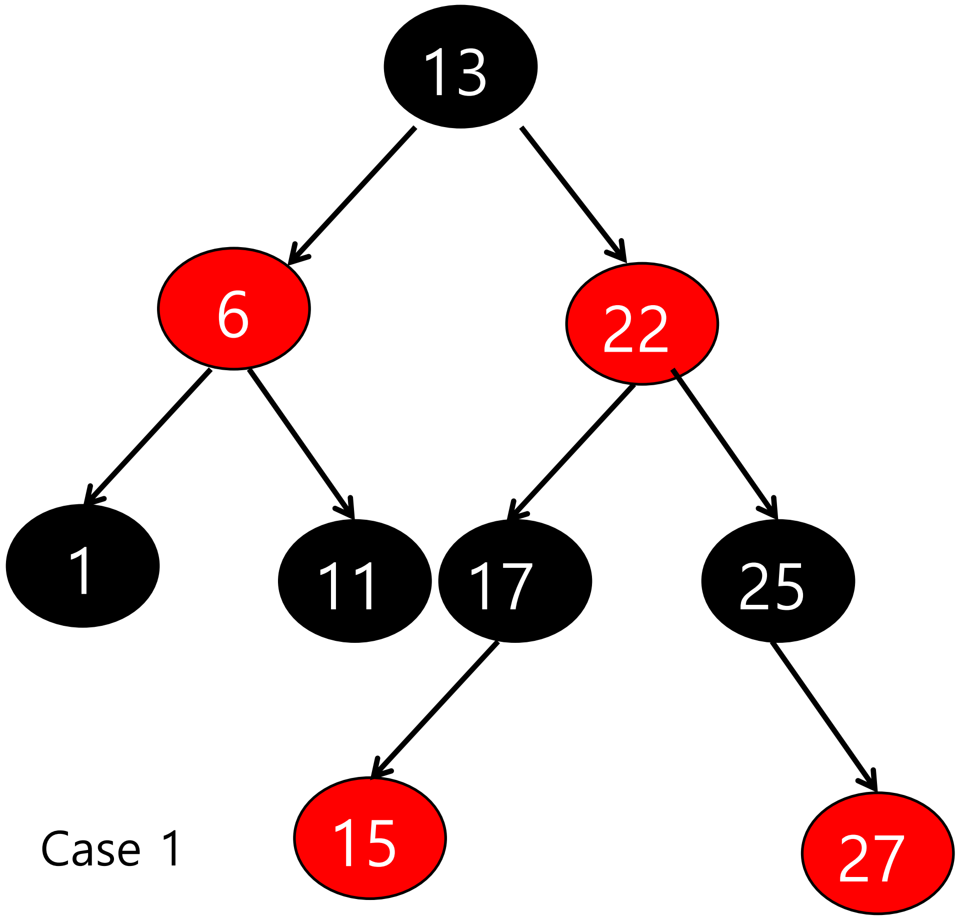
Example: Delete 17



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

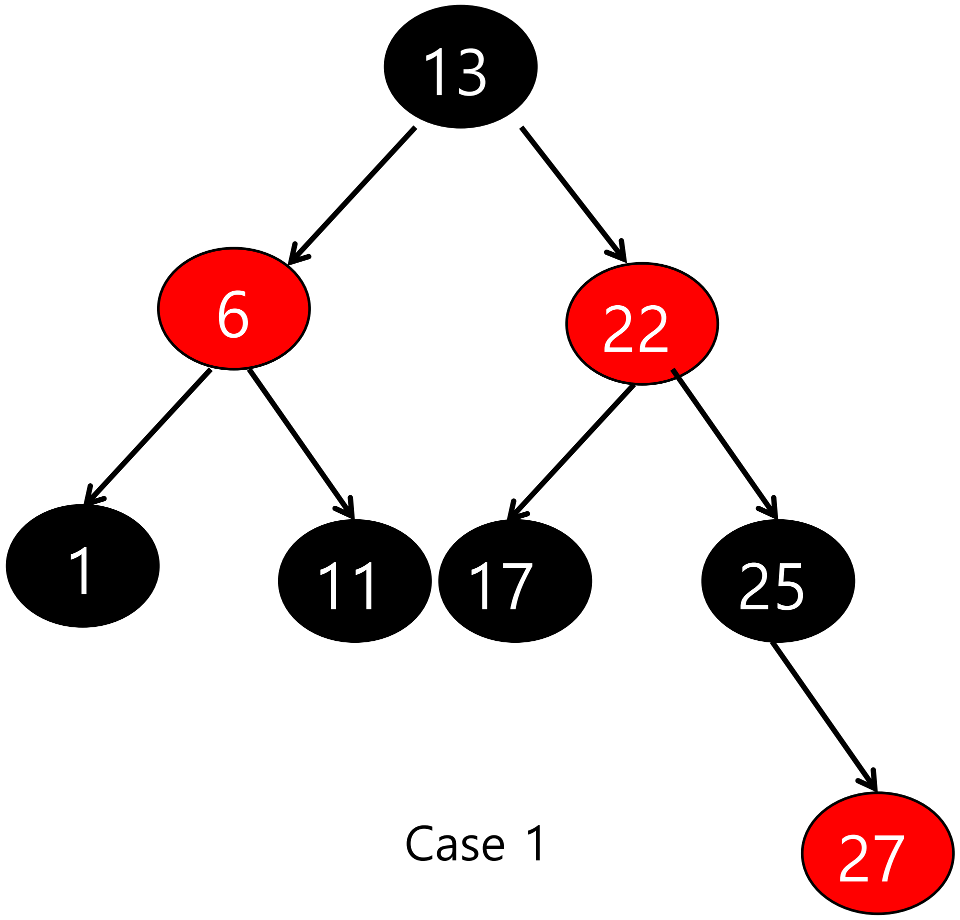


Exercise: Delete 8



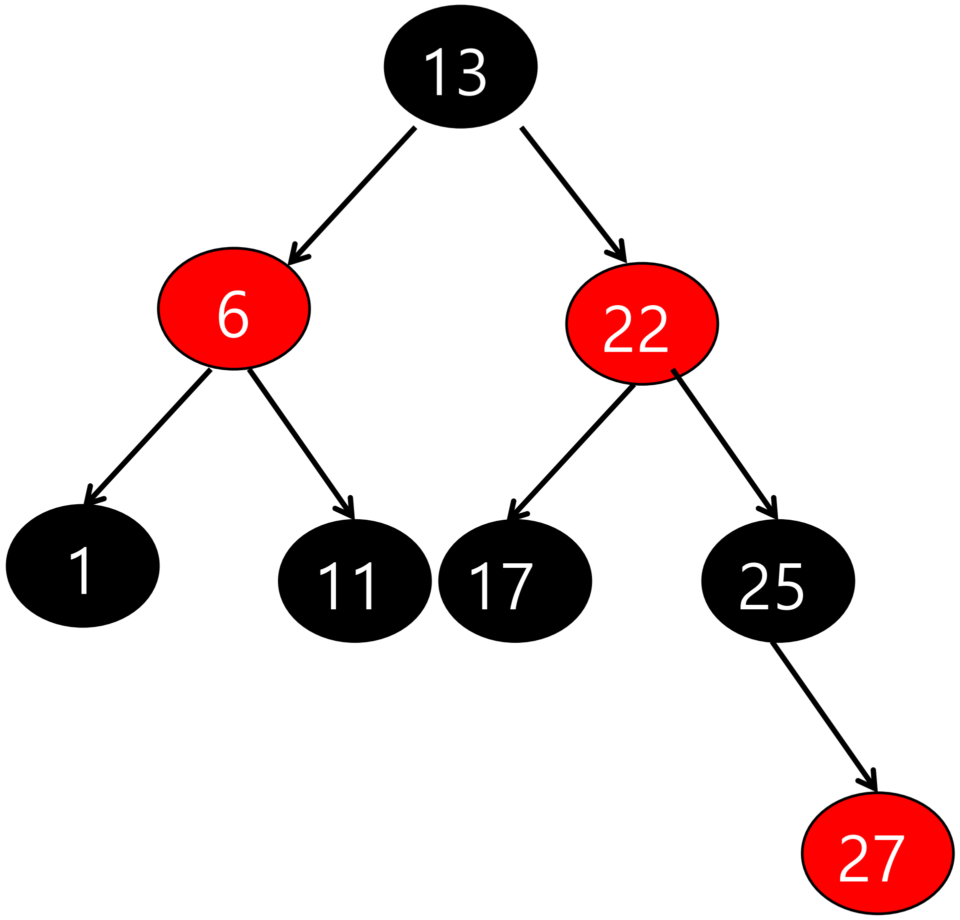
Case 1

Exercise: Delete 15

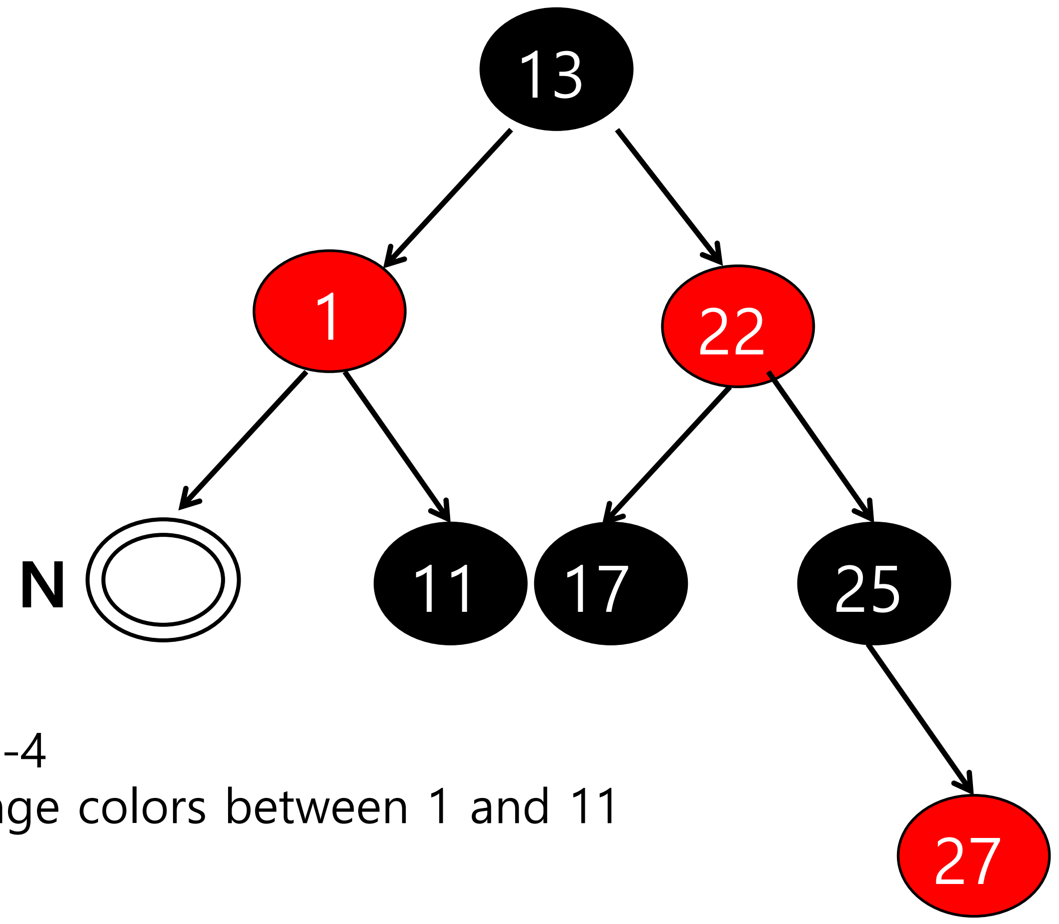


Case 1

Exercise: Delete 6

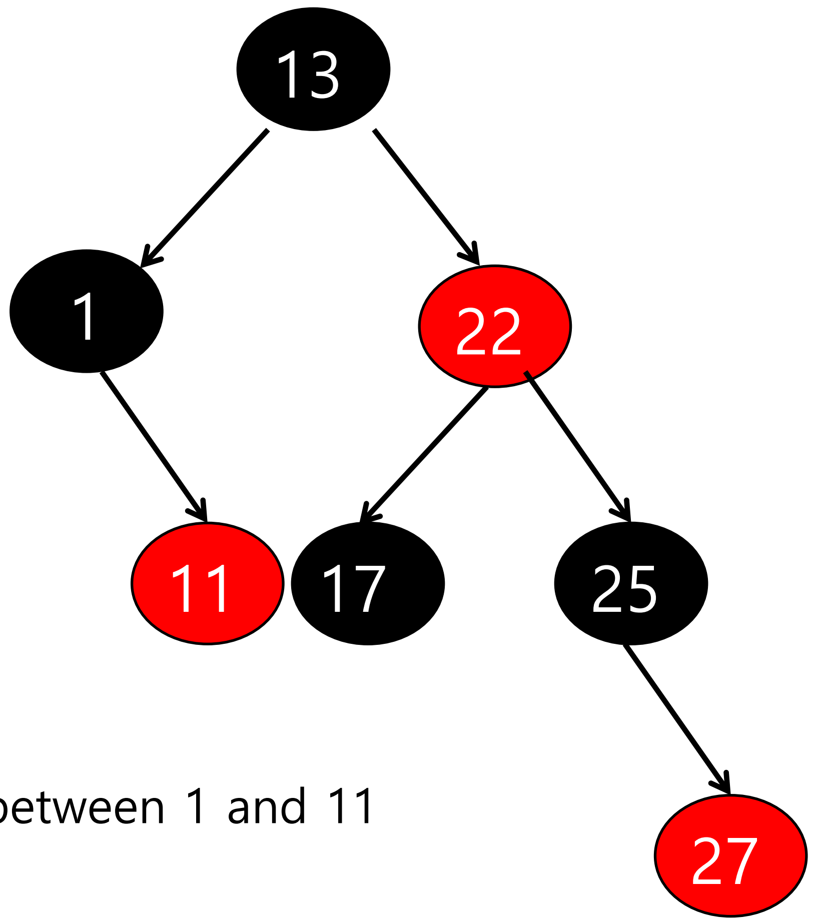


Exercise: Delete 6



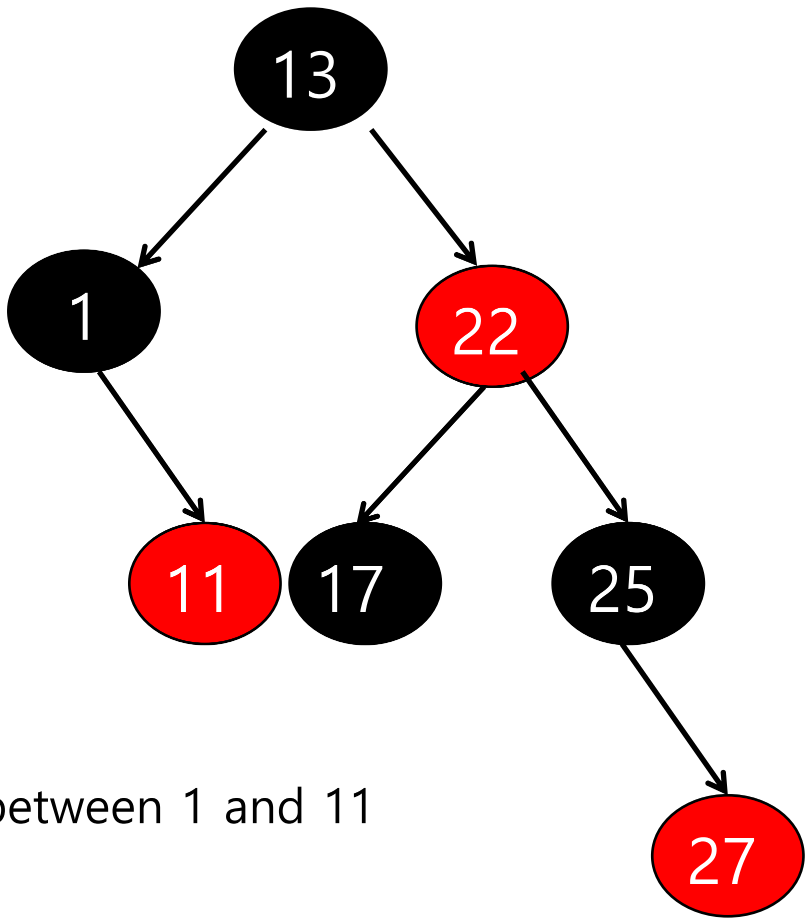
Case 3-4
Exchange colors between 1 and 11

Exercise: Delete 6



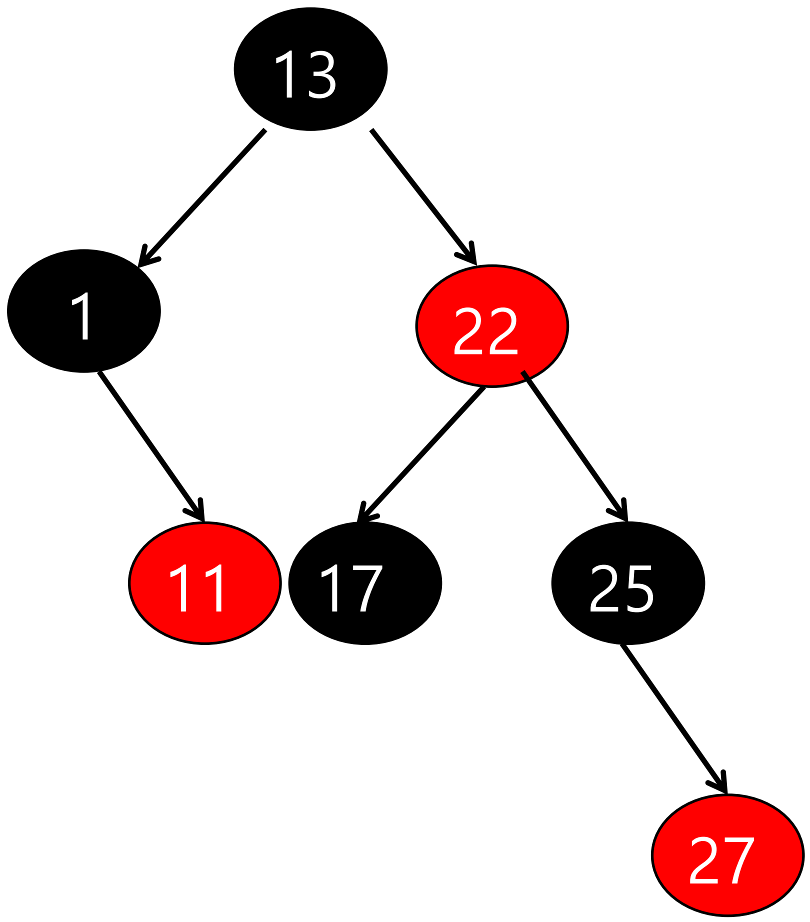
Case 3-4
Exchange colors between 1 and 11

Exercise: Delete 13

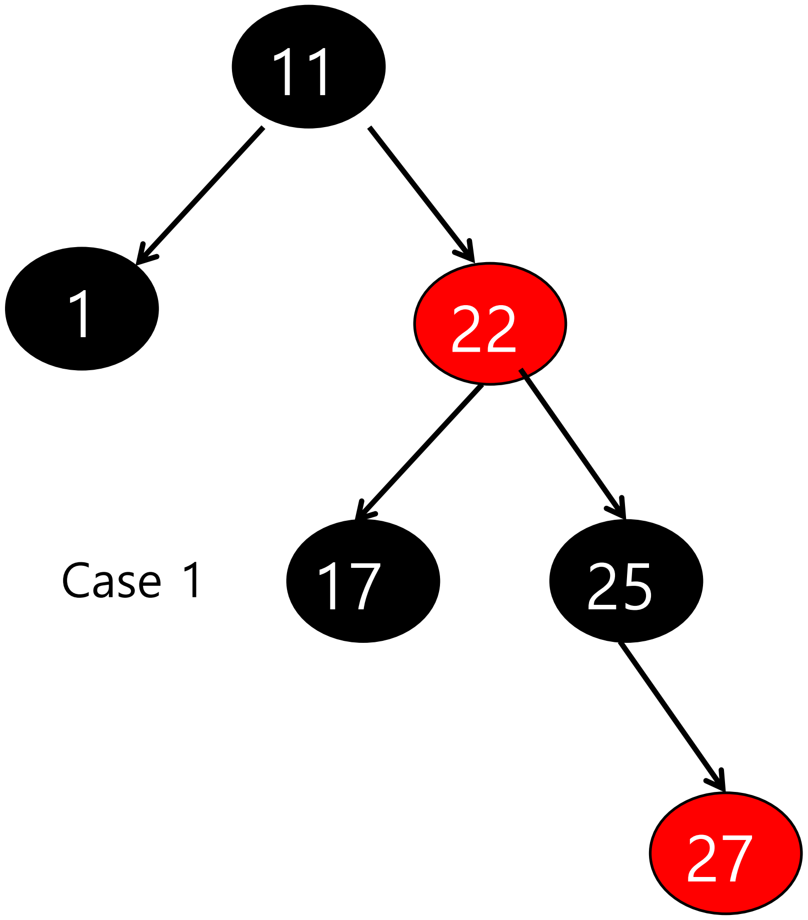


Case 3-4
Exchange colors between 1 and 11

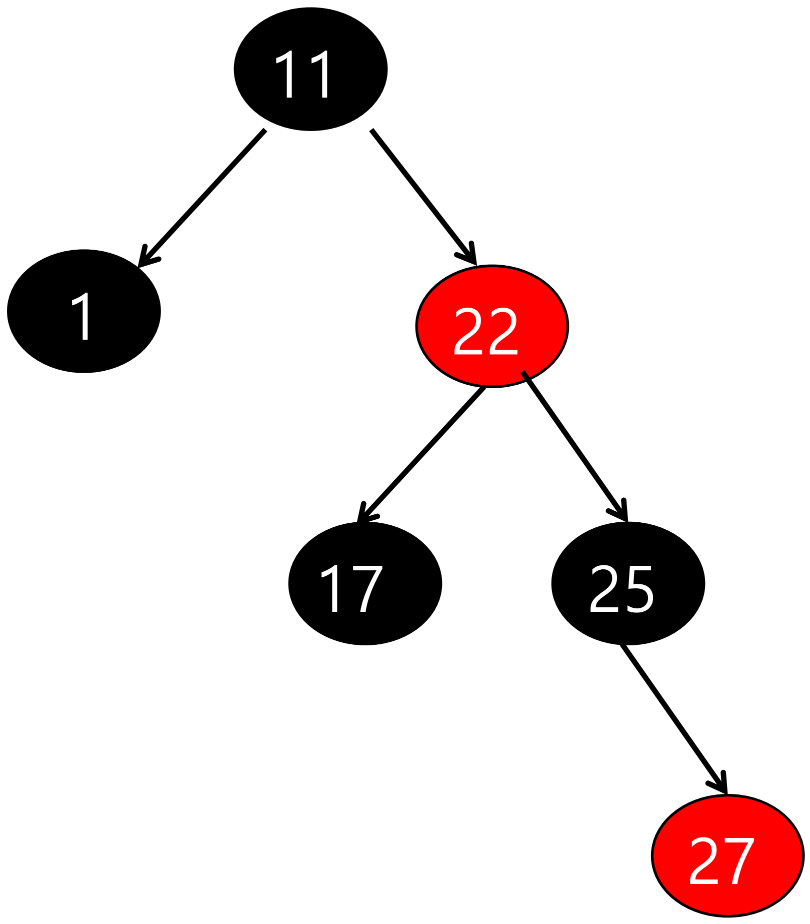
Exercise: Delete 13



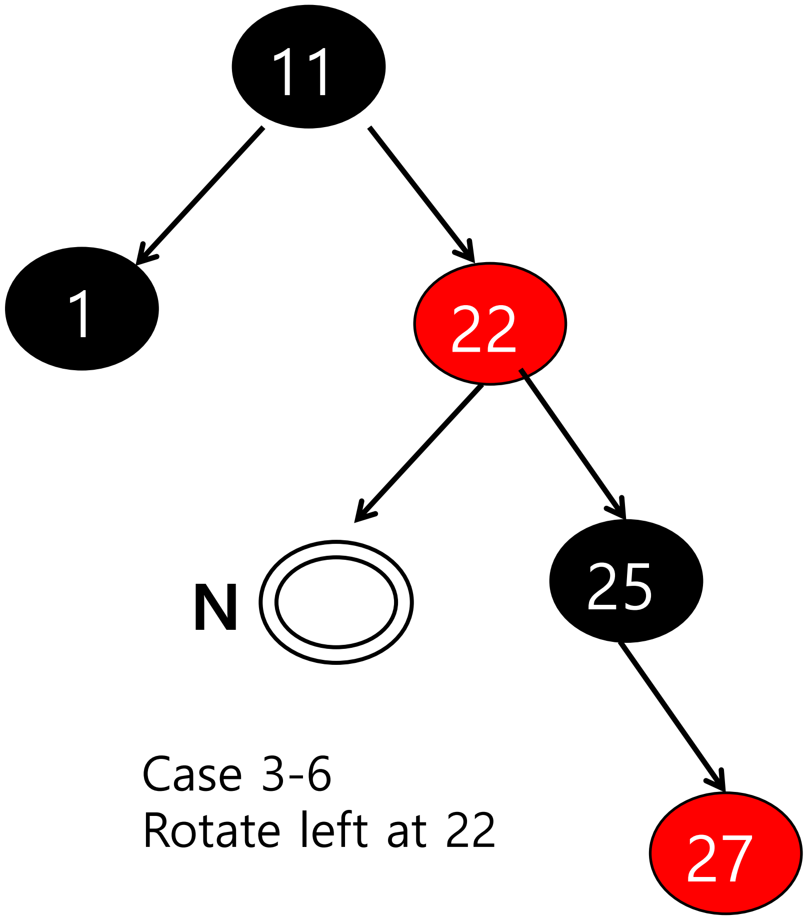
Exercise: Delete 13



Exercise: Delete 17

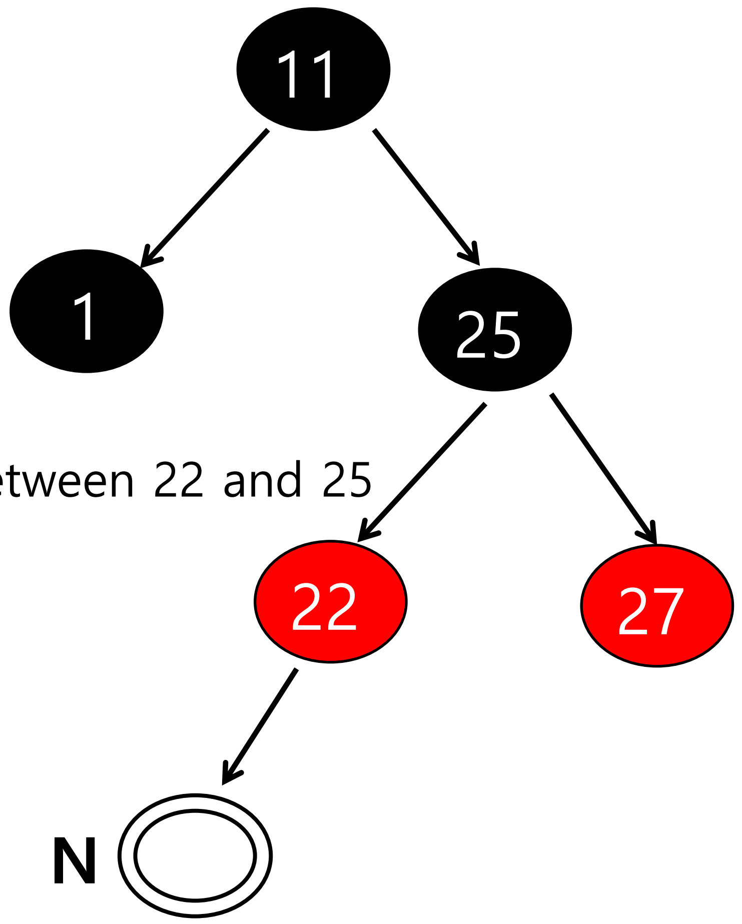


Exercise: Delete 17



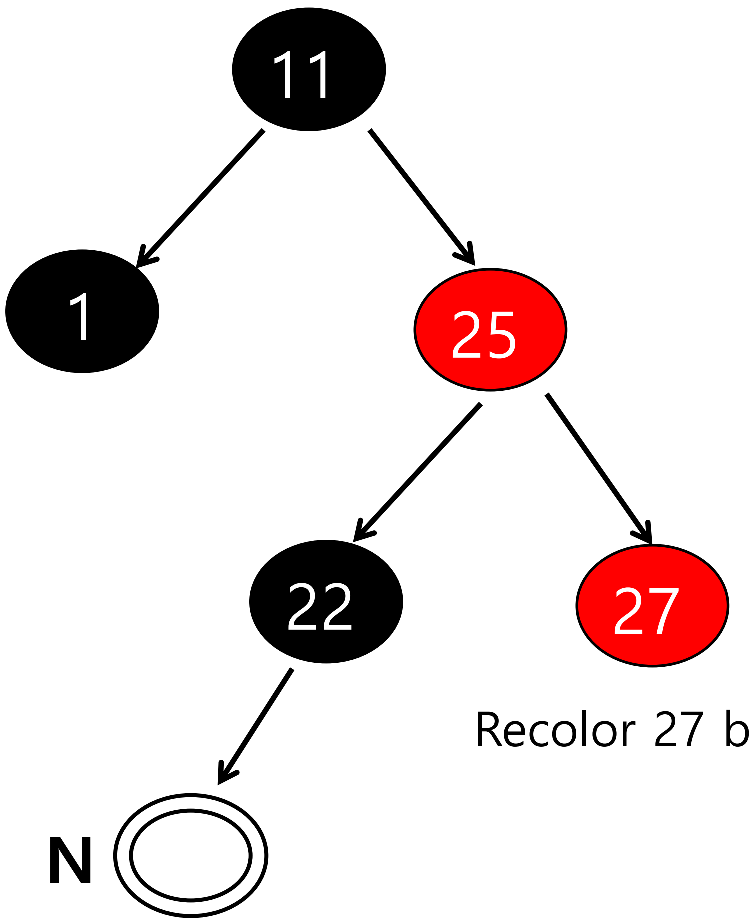
Case 3-6
Rotate left at 22

Exercise: Delete 17



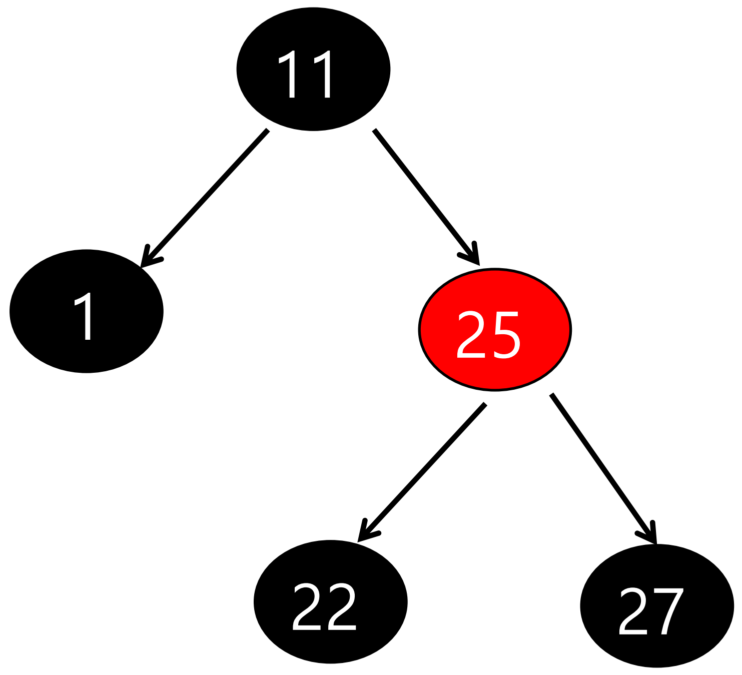
Exchange colors between 22 and 25

Exercise: Delete 17



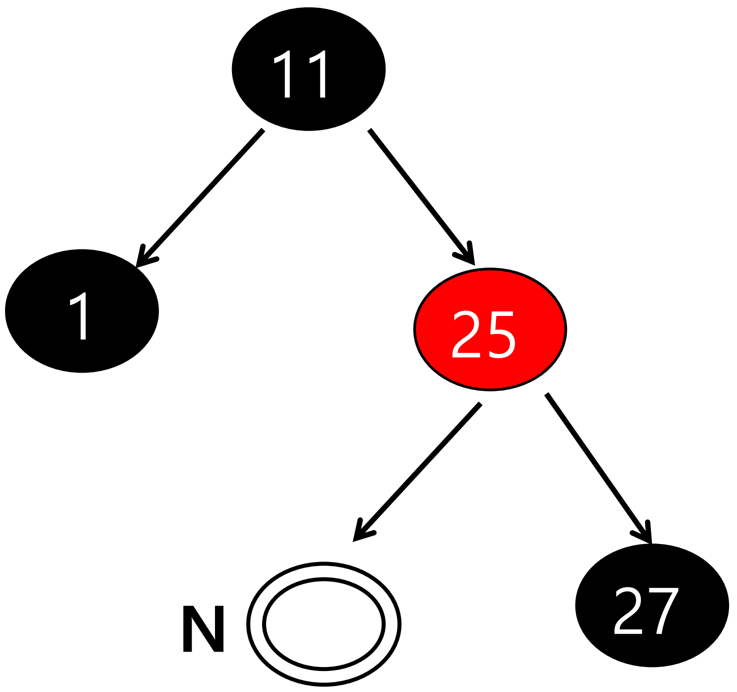
Recolor 27 black

Exercise: Delete 17



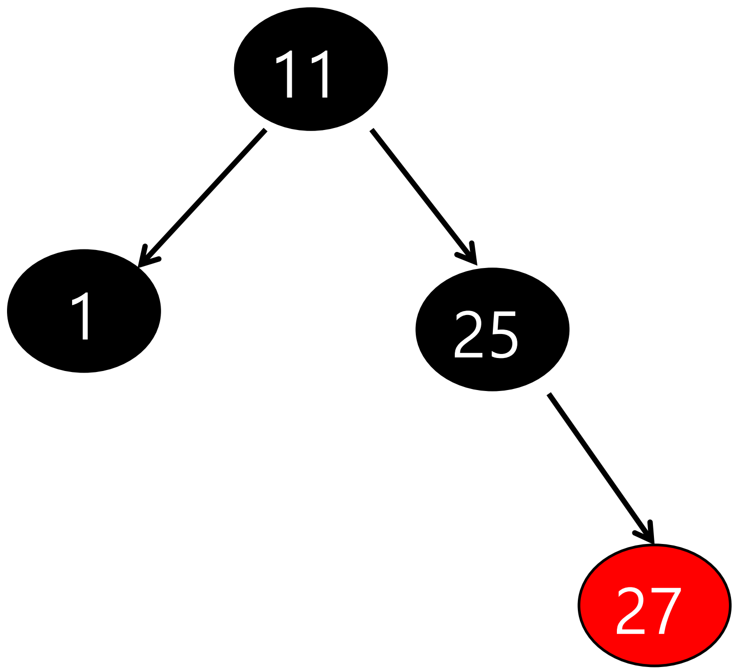
Recolor 27 black

Exercise: Delete 22



Case 3-4
Exchange colors between 25 and 27

Exercise: Delete 22



Deletion Summary

- A red node can be simply deleted (Case 1).
- A black node becomes **double black** when deleted.
- Double black needs to be removed to preserve RB-tree properties
  - Child is red: Replace and color it black (Case 2).
  - Child is black (Case 3).
    - Case 3-1 (T): **N** is a new root
    - Case 3-2: **S**
    - Case 3-3: **S, P, S_L, S_R**
    - Case 3-4 (T): **S, P, S_L, S_R**
    - Case 3-5: **S, S_L, S_R**
    - Case 3-6 (T): **S, S_L, S_R**

* **S_L** and **S_R** represents close and distant child, respectively (NOT necessarily indicating left and right)

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.

WHW 4 (20 points) 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 Class