# Red Black Trees

# Question 1

- 2000 elements are inserted one at a time into an initially empty binary search tree using the traditional algorithm. What is the maximum possible height of the resulting tree?

A. 1

B. 11

C. 1000

D. 1999

E. 4000

# Binary Search Trees

- Average case and worst case Big O for
  - insertion
  - deletion
  - access
- Balance is important. Unbalanced trees give worse than log N times for the basic tree operations
- Can balance be guaranteed?

# Red Black Trees

- A BST with more complex algorithms to ensure balance
- Each node is labeled as Red or Black.
- Path: A unique series of links (edges) traverses from the root to each node.
  - The number of edges (links) that must be followed is the path length
- In Red Black trees paths from the root to elements with 0 or 1 child are of particular interest

# Red Black Trees

## Colored Nodes Definition

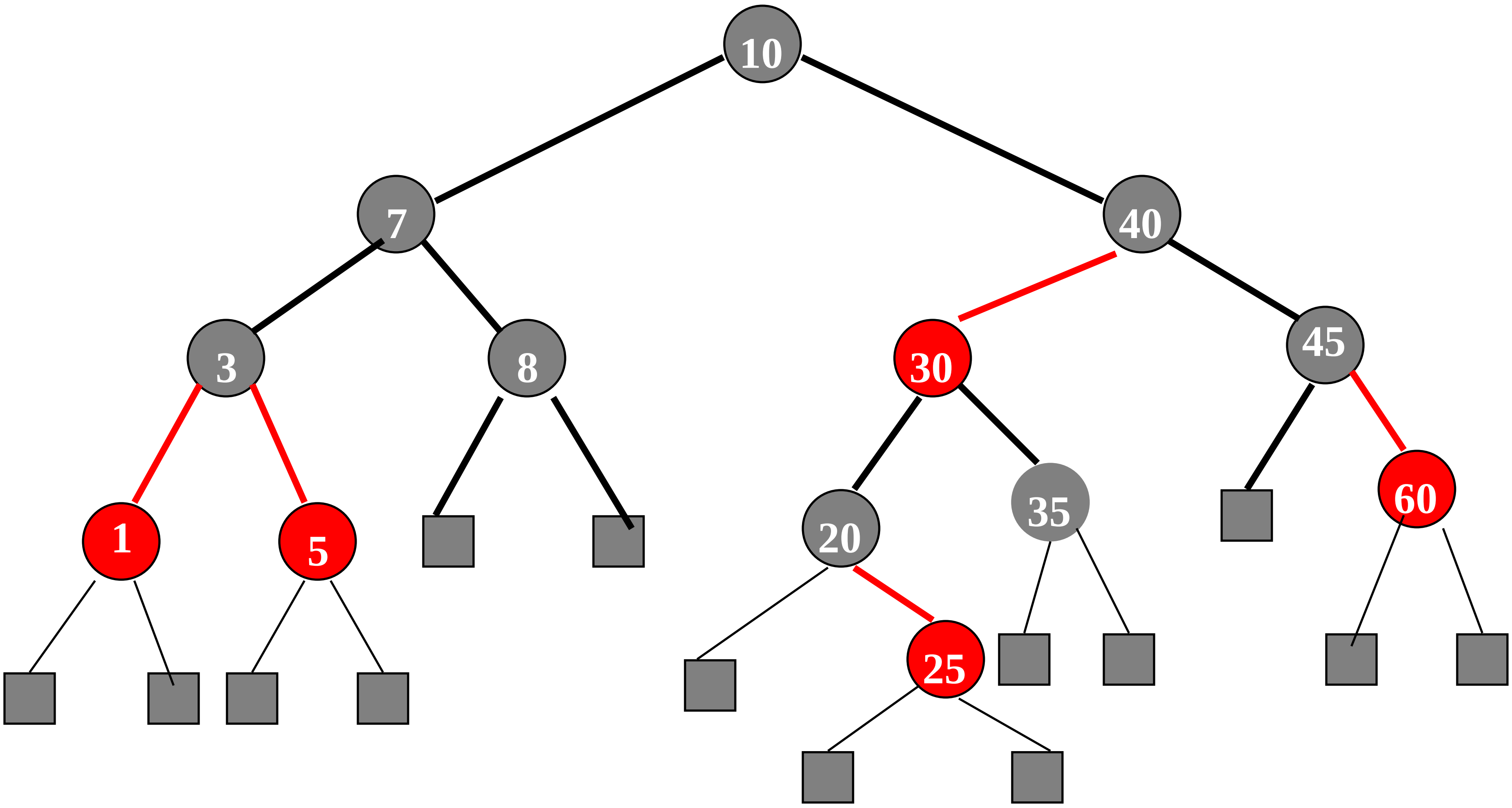
- Binary search tree.
- Each node is colored red or black.
- Root and all external nodes are black.
- No root-to-external-node path has two consecutive red nodes.
- All root-to-external-node paths have the same number of black nodes

# Red Black Trees

## Colored Edges Definition

- Binary search tree.
- Child pointers are colored red or black.
- Pointer to an external node is black.
- No root to external node path has two consecutive red pointers.
- Every root to external node path has the same number of black pointers.

# Example Red-Black Tree



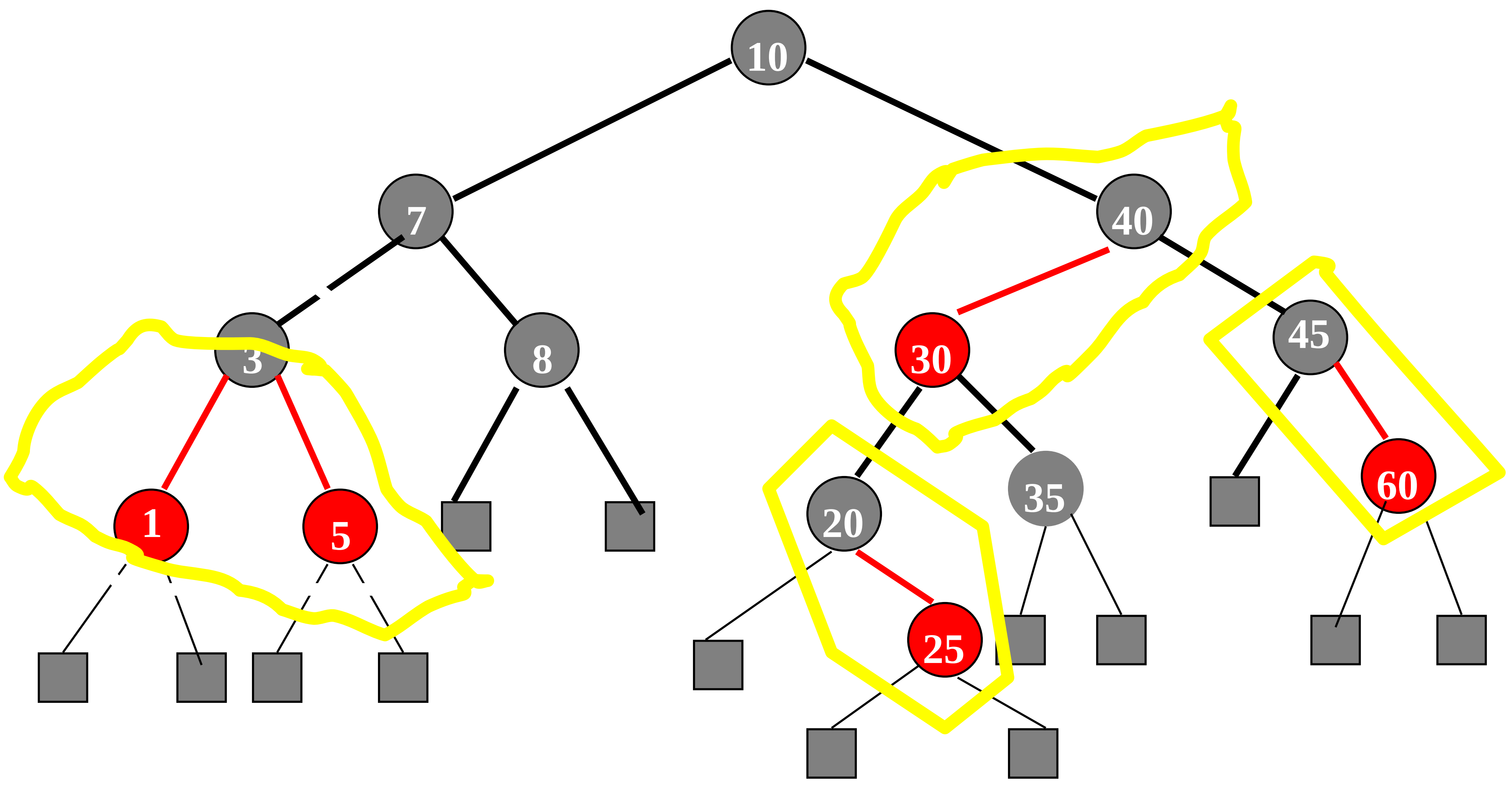
# Properties

- The height of a red black tree that has n (internal) nodes is between $\log_2(n+1)$ and $2\log_2(n+1)$.

# Properties

- Start with a red black tree whose height is h; collapse all red nodes into their parent black nodes to get a tree whose node-degrees are between 2 and 4, height is >= h/2, and all external nodes are at the same level.

# Properties

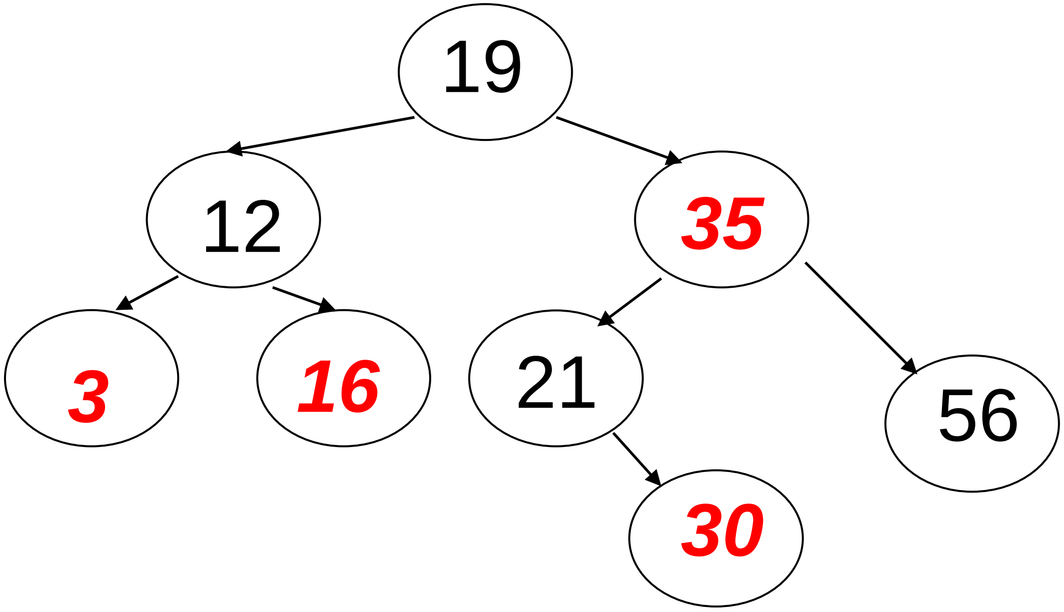


# Properties

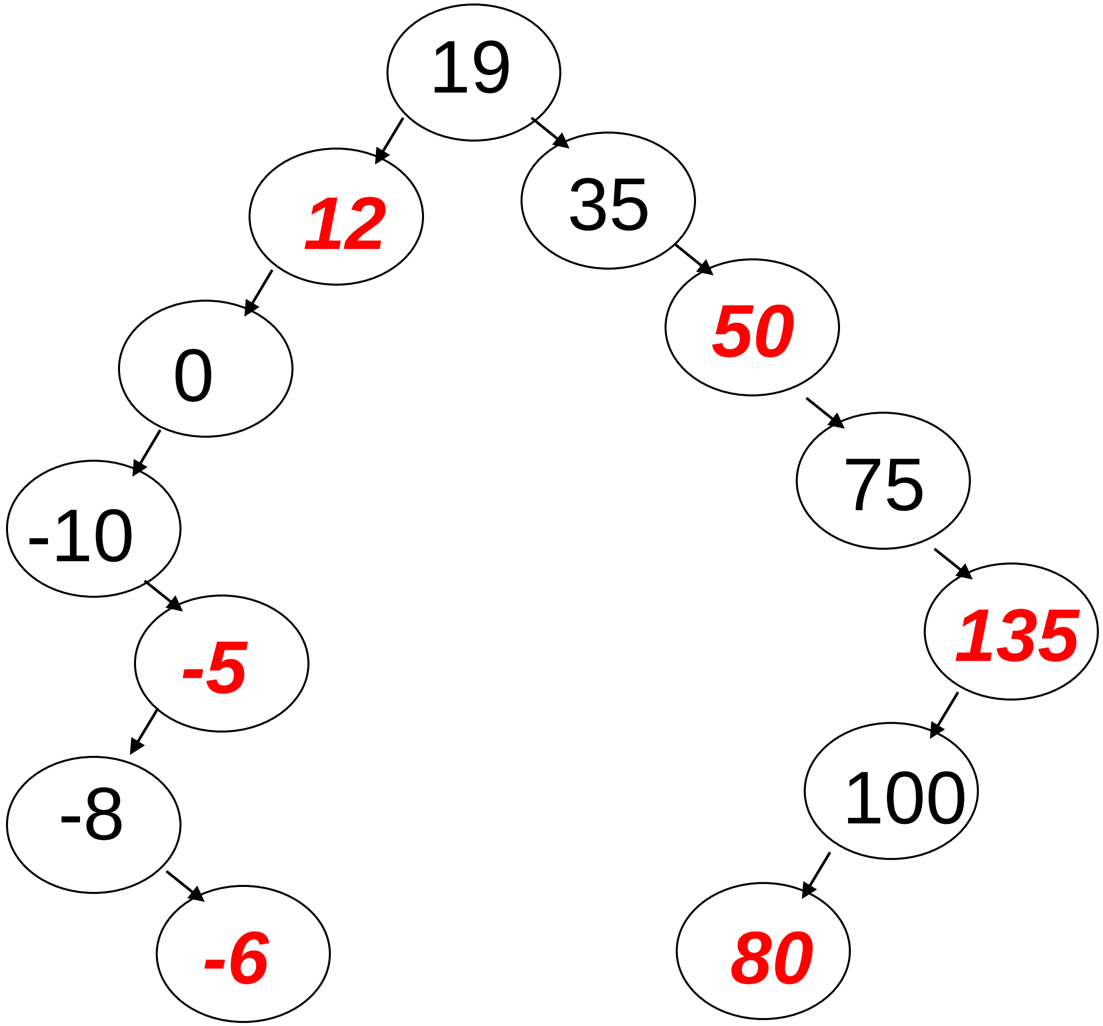
- Let $h' \geq h/2$ be the height of the collapsed tree.
- Internal nodes of collapsed tree have degree between 2 and 4.
- Number of internal nodes in collapsed tree $\geq 2^{h'}-1$.
- So, $n \geq 2^{h'}-1$
- So, $h \leq 2 \log_2 (n + 1)$

# Example of a Red Black Tree

- The root of a Red Black tree is black
- Every other node in the tree follows these rules:
  - If a node is Red, all of its children are Black
  - The number of Black nodes must be the same in all paths from the root node to null nodes



# Red Black Tree?

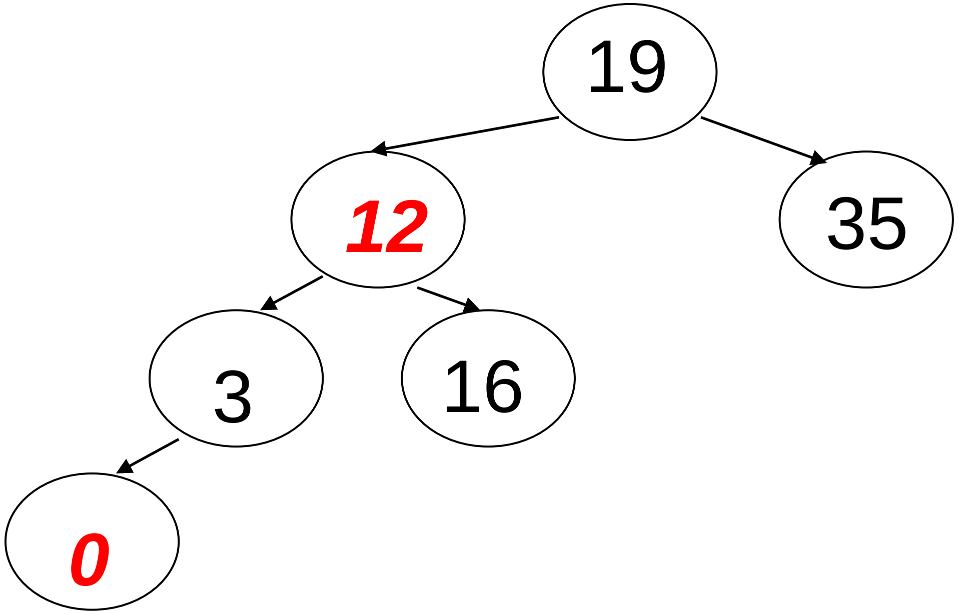


# Question 2

- Is the tree on the previous slide a binary search tree? Is it a red black tree?

| | BST? | Red-Black? |
|---|---|---|
| A. | No | No |
| B. | No | Yes |
| C. | Yes | No |
| D. | Yes | Yes |

# Red Black Tree?




Perfect?

Full?

Complete?

# Question 3

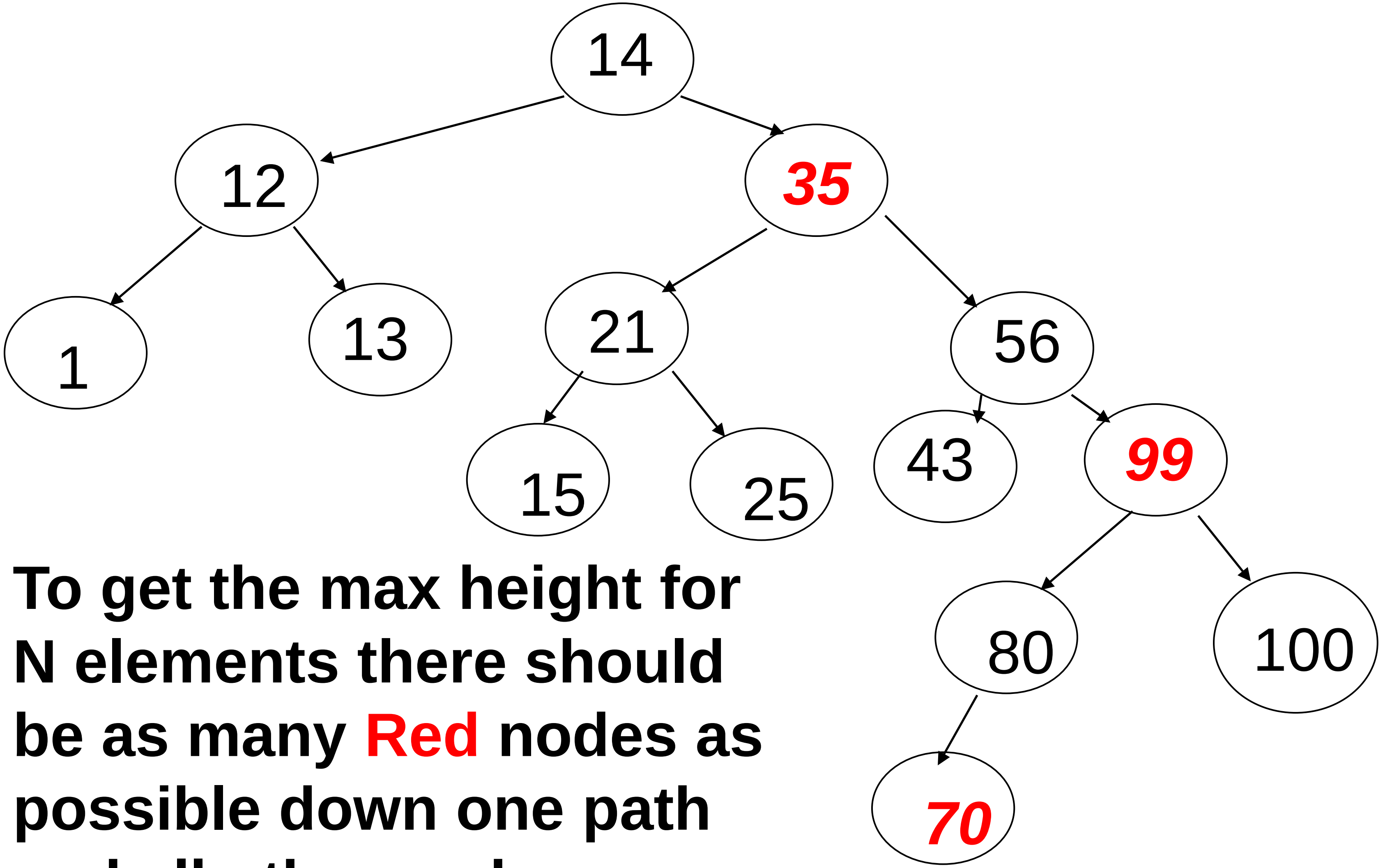
- Is the tree on the previous slide a binary search tree? Is it a red black tree?

| | BST? | Red-Black? |
|---|---|---|
| A. | No | No |
| B. | No | Yes |
| C. | Yes | No |
| D. | Yes | Yes |

# Implications of the Rules

- If a Red node has any children, it must have two children and they must be Black. (Why?)
- If a Black node has only one child that child must be a Red leaf. (Why?)
- Due to the rules there are limits on how unbalanced a Red Black tree may become.
  - on the previous example may we hang a new node off of the leaf node that contains 0?

# Max Height Red Black Tree




**To get the max height for N elements there should be as many Red nodes as possible down one path and all other nodes are Black**

# Maintaining the Red Black Properties in a Tree

- Insertions
- Must maintain rules of Red Black Tree.
- New Node always a leaf
  - can't be black or we will violate a rule
  - therefore the new leaf must be red
  - If parent is black, done (trivial case)
  - if parent red, things get interesting because a red leaf with a red parent violates a rule

# Bottom-Up Rebalancing for Red-Black Trees

*** The idea for insertion in a red-black tree is to insert like in a binary search tree and then reestablish the color properties through a sequence of recoloring and rotations**

**The rules are as follows:**

**1. If other is red, color current and other black and upper red.**

**2. If current = upper->left**

**2.1 If current->right->color is black,**

**perform a right rotation around upper and color upper->right red.**

**2.2 If current->right->color is red,**

**perform a left rotation around current followed by a right rotation around upper, and color upper->right and upper->left black and upper red.**

**3. If current = upper->right**

**3.1 If current->left->color is black,**

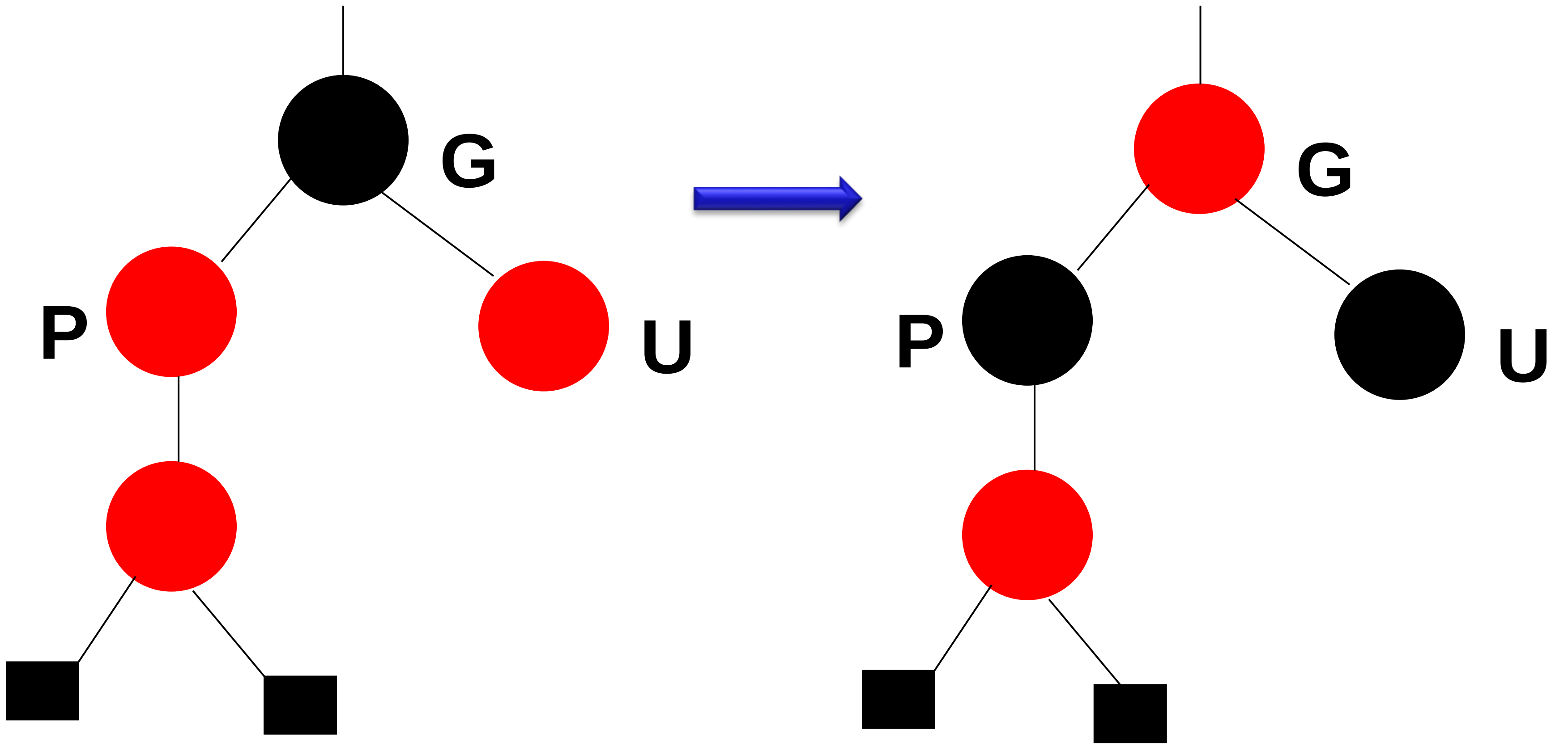
**perform a left rotation around upper and color upper->left red.**

**3.2 If current->left->color is red,**

**perform a right rotation around current followed by a left rotation around upper, and color upper->right and upper->left black and upper red.**

* We have 3 cases for insertion

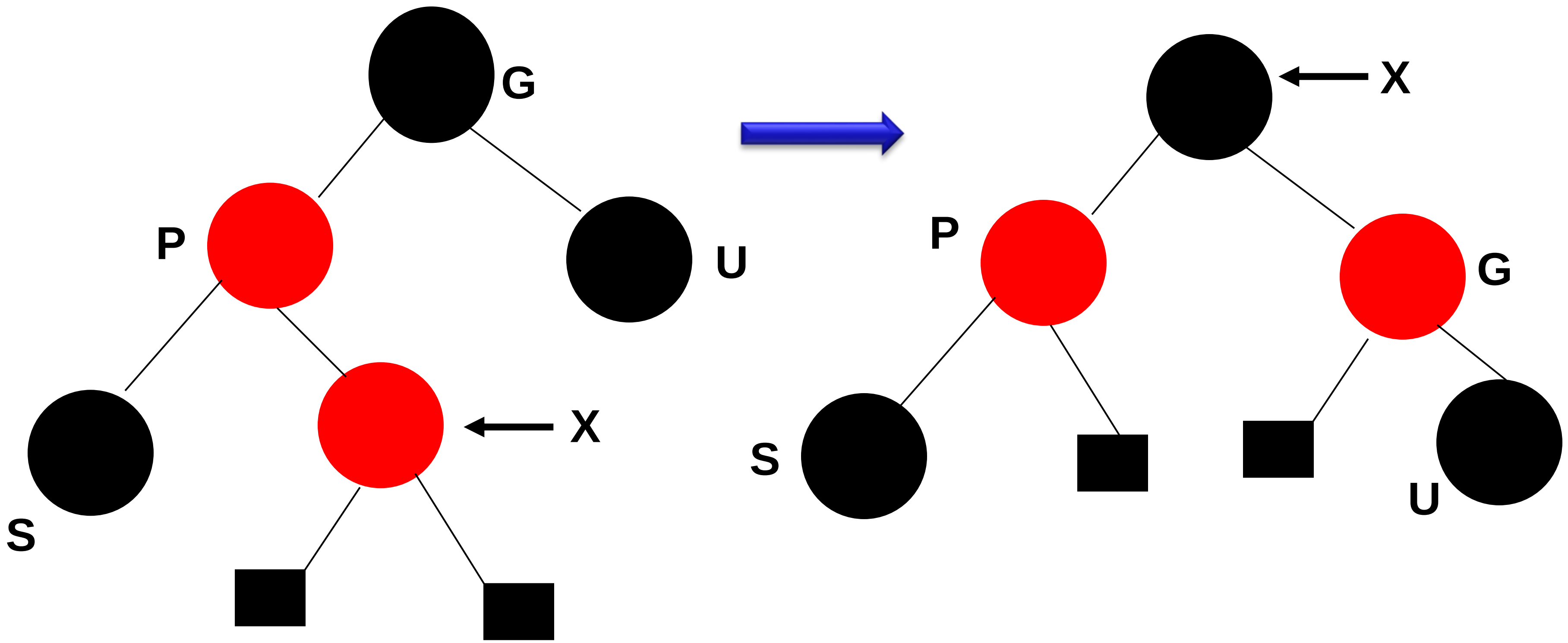
## Case 1: Recolor (uncle is red)



**Case 2:**

**Double Rotate: X around P then X around G.**

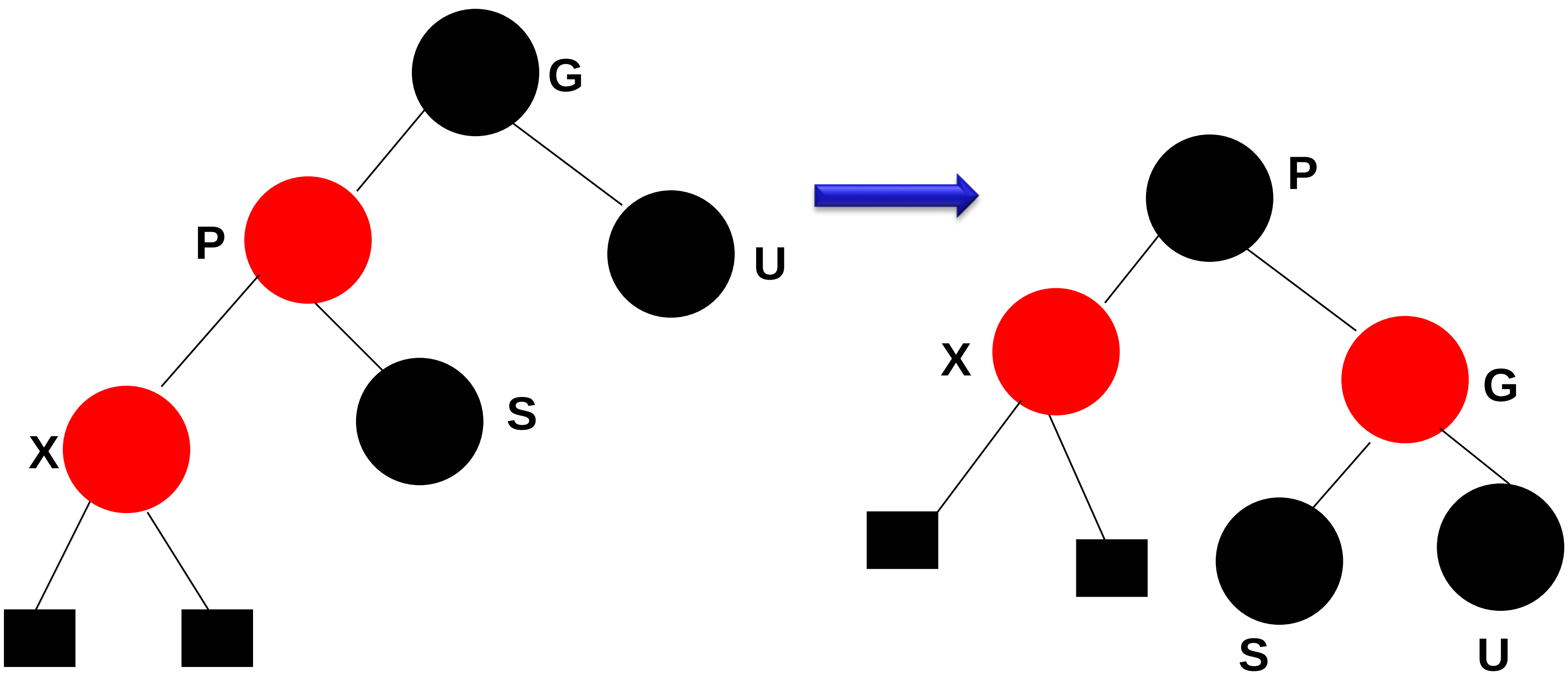
**Recolor G and X**



Case 3:

Single Rotate P around G

Recolor P and G



# Analysis of Insertion

- **A red-black tree has *O*(log *n*) height**
- **Search for insertion location takes *O*(log *n*) time because we visit *O*(log *n*) nodes**
- **Addition to the node takes *O*(1) time**
- **Rotation or recoloring takes *O*(log *n*) time because we perform**

*** *O*(log *n*) recoloring, each taking *O*(1) time, and**

*** at most one rotation taking *O*(1) time**

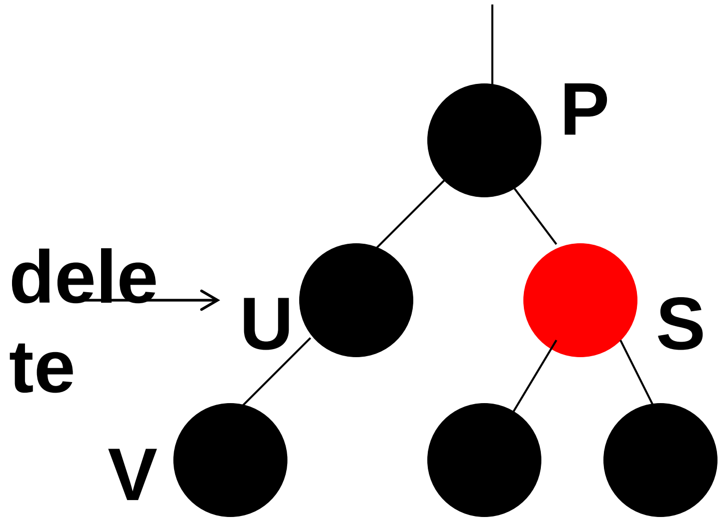
**- Thus, an insertion in a red-black tree takes *O*(log *n*) time**

- **Deleting a node from a red-black tree is a bit more complicated than inserting a node.**

- **If the node is red?**
  **Not a problem – no RB properties violated**

- **If the node is black?**
  **deleting it will change the black-height along some path**

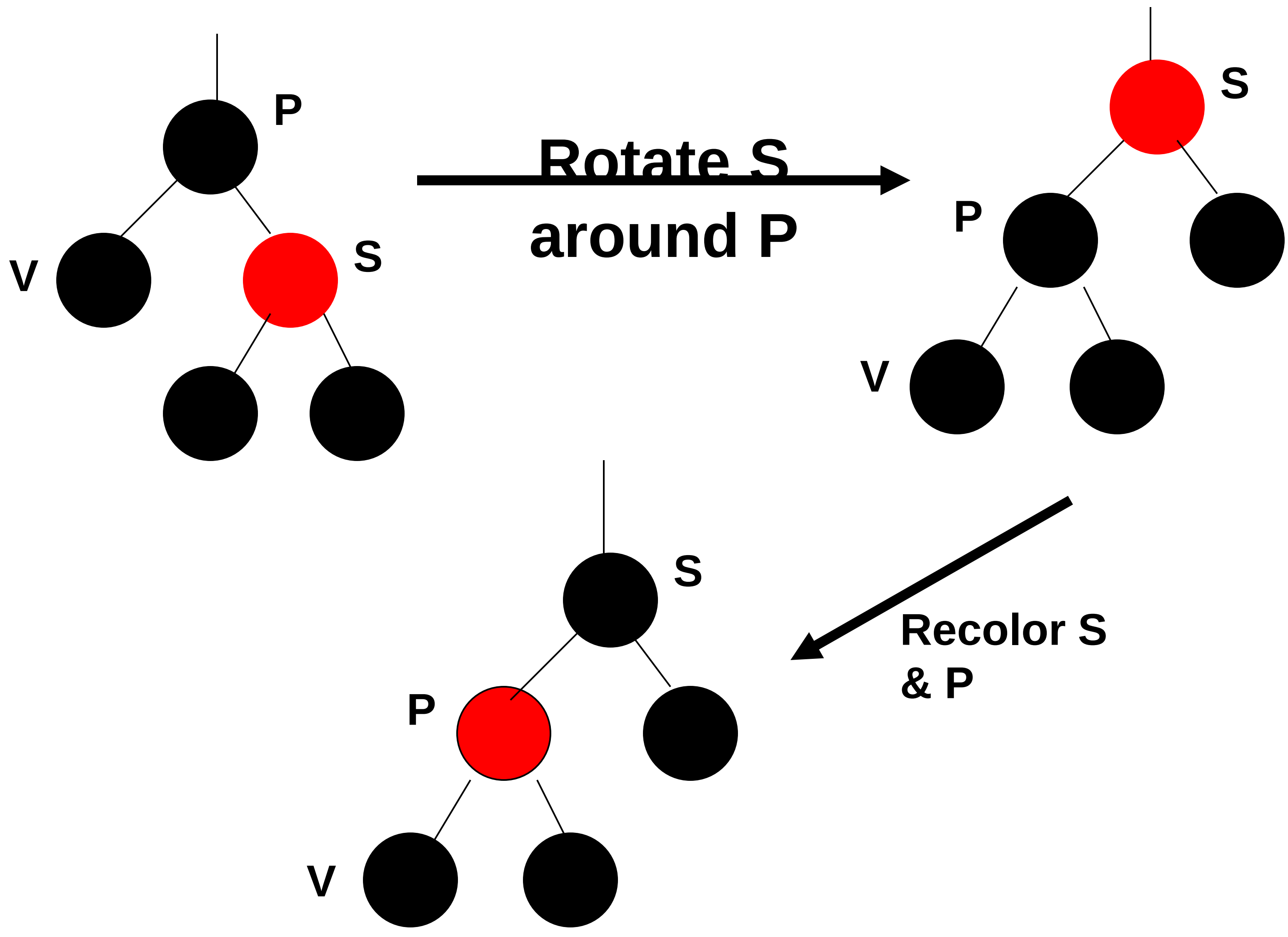
* We have some cases for deletion

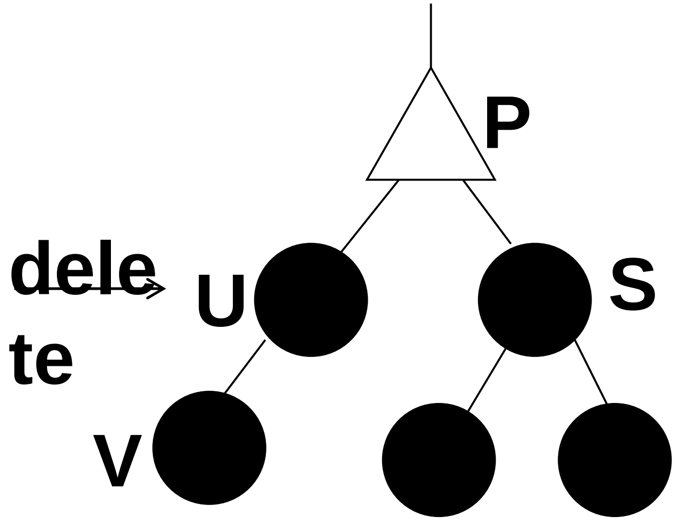



Case A:

- V’s sibling, S, is Red

  Rotate S around P and recolor S & P

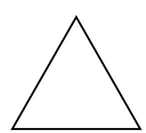

P
V
S
Rotate S
around P
S
P
V
Recolor S
& P
S
P
V



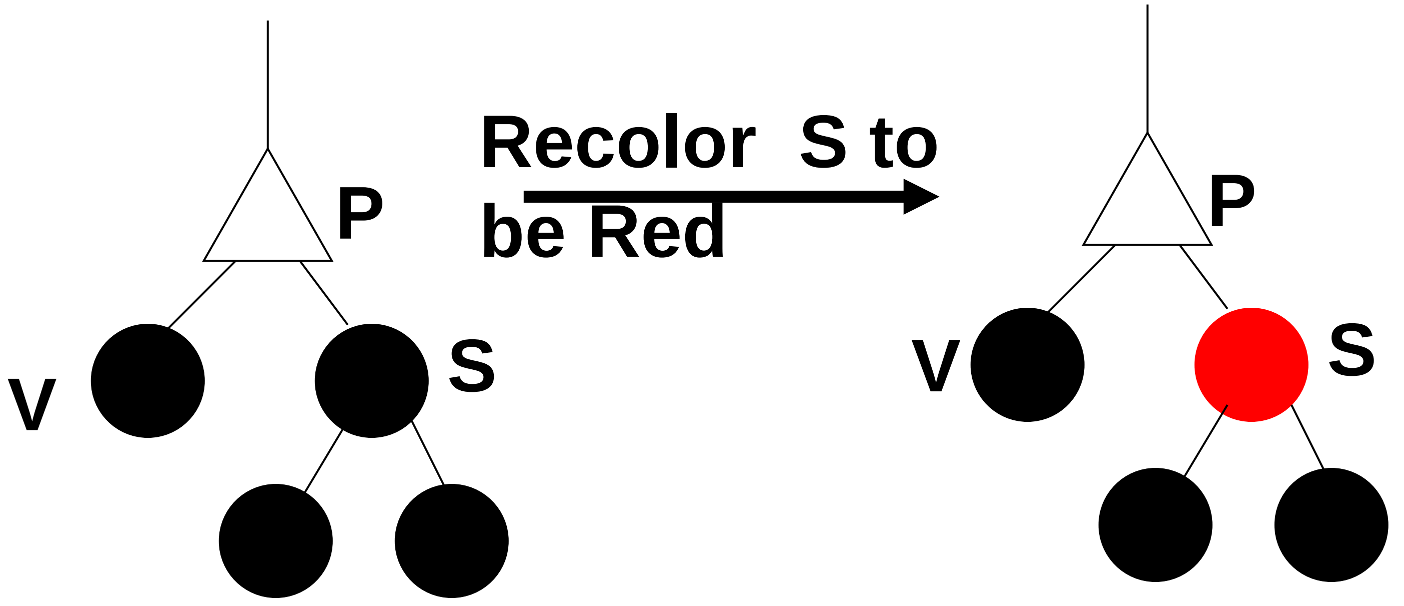

**Case B:**

**- V's sibling, S, is black and has two black children.**

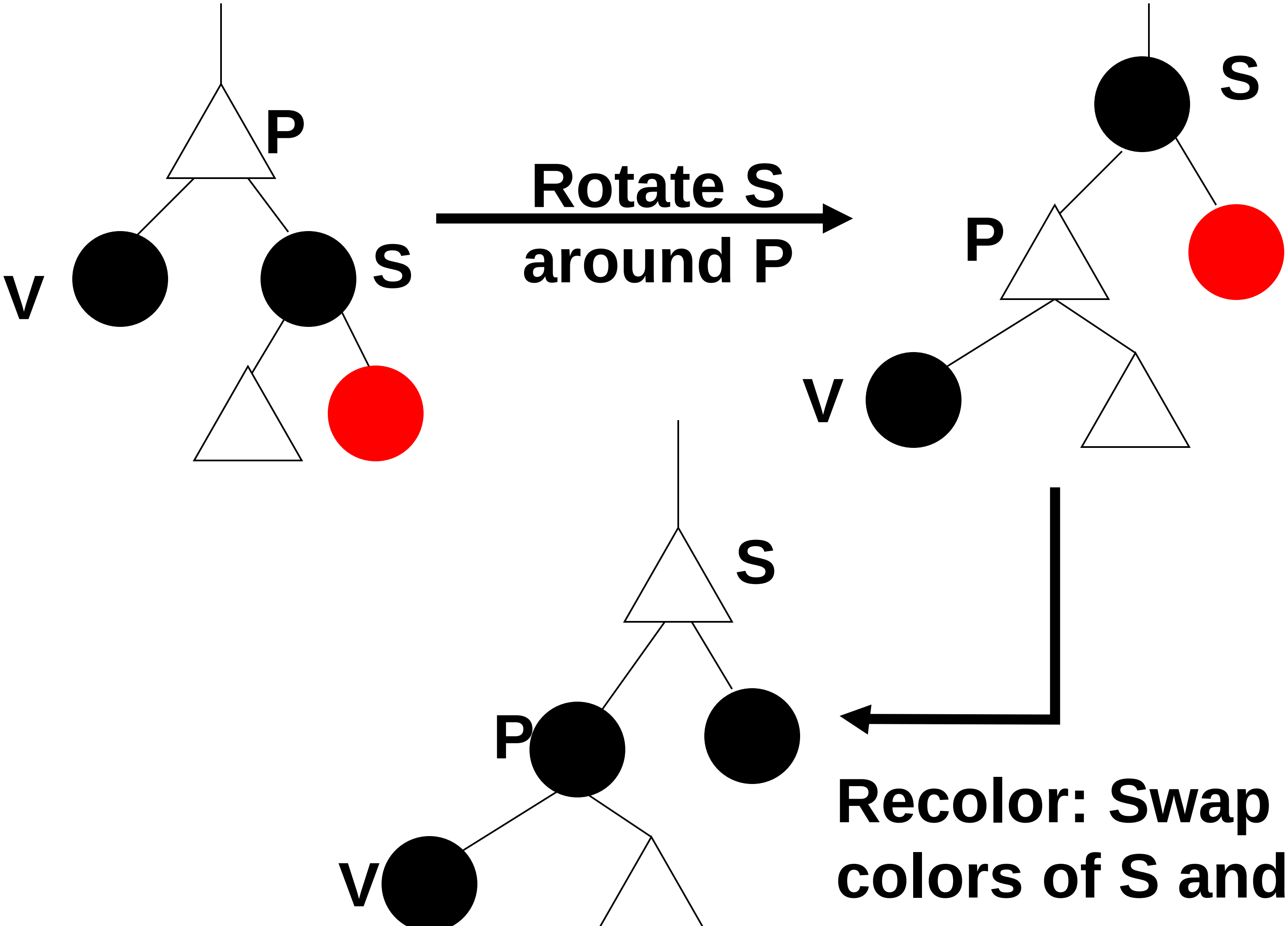
**Recolor S to be Red**



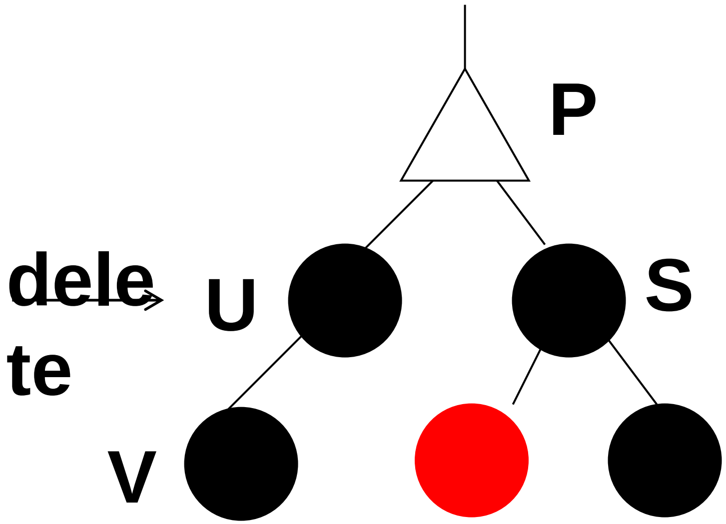
**Red or Black and don't care**



P
V
S
Recolor S to
be Red
P
V
S




**Recolor: Swap colors of S and P, and color S's**

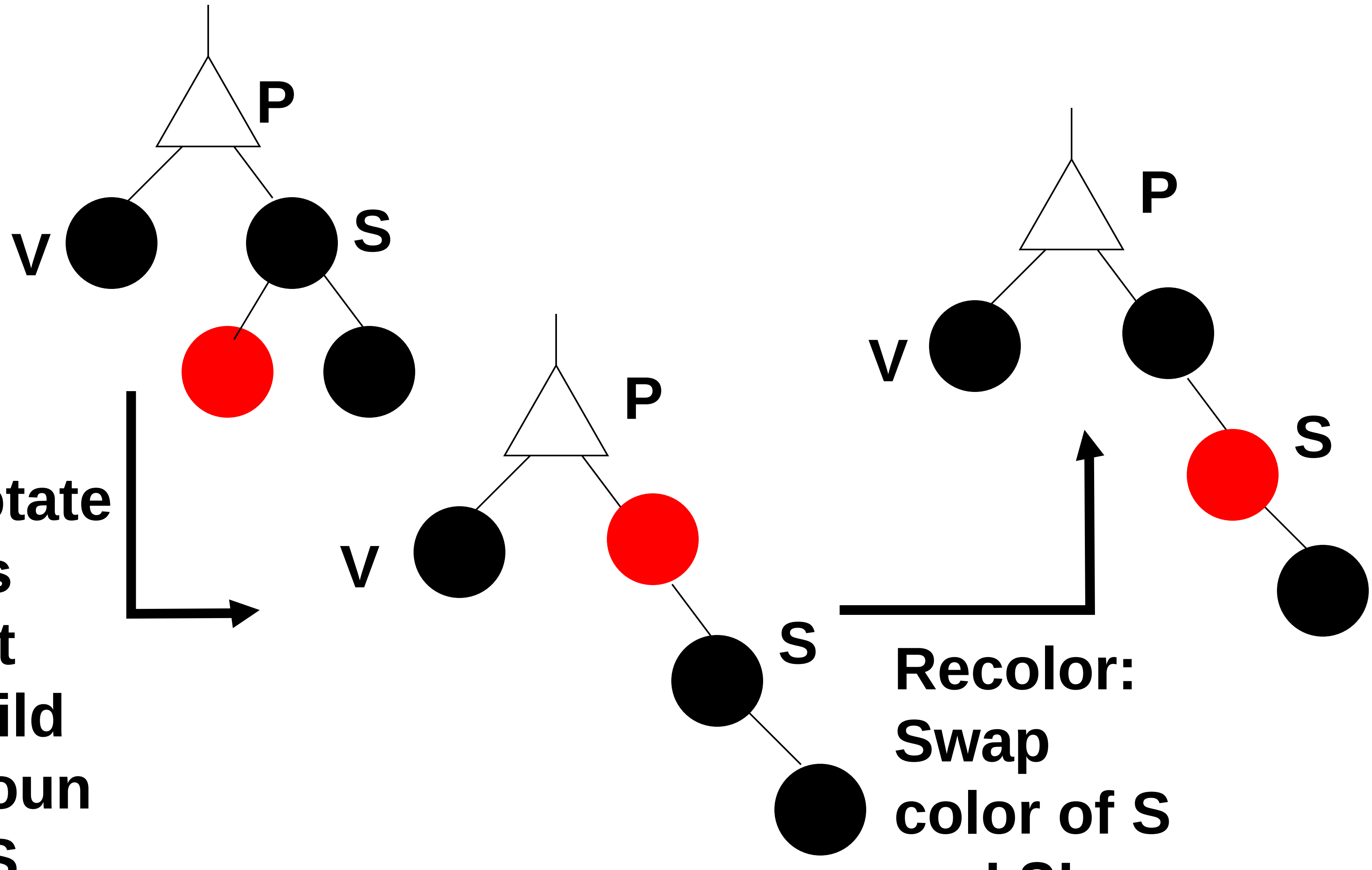



**Case D:**

**- S is Black, S's right child is Black and S's left child is Red**

**i) Rotate S's left child around S**

**ii) Swap color of S and S's left child**




**Rotate S's left child around S**

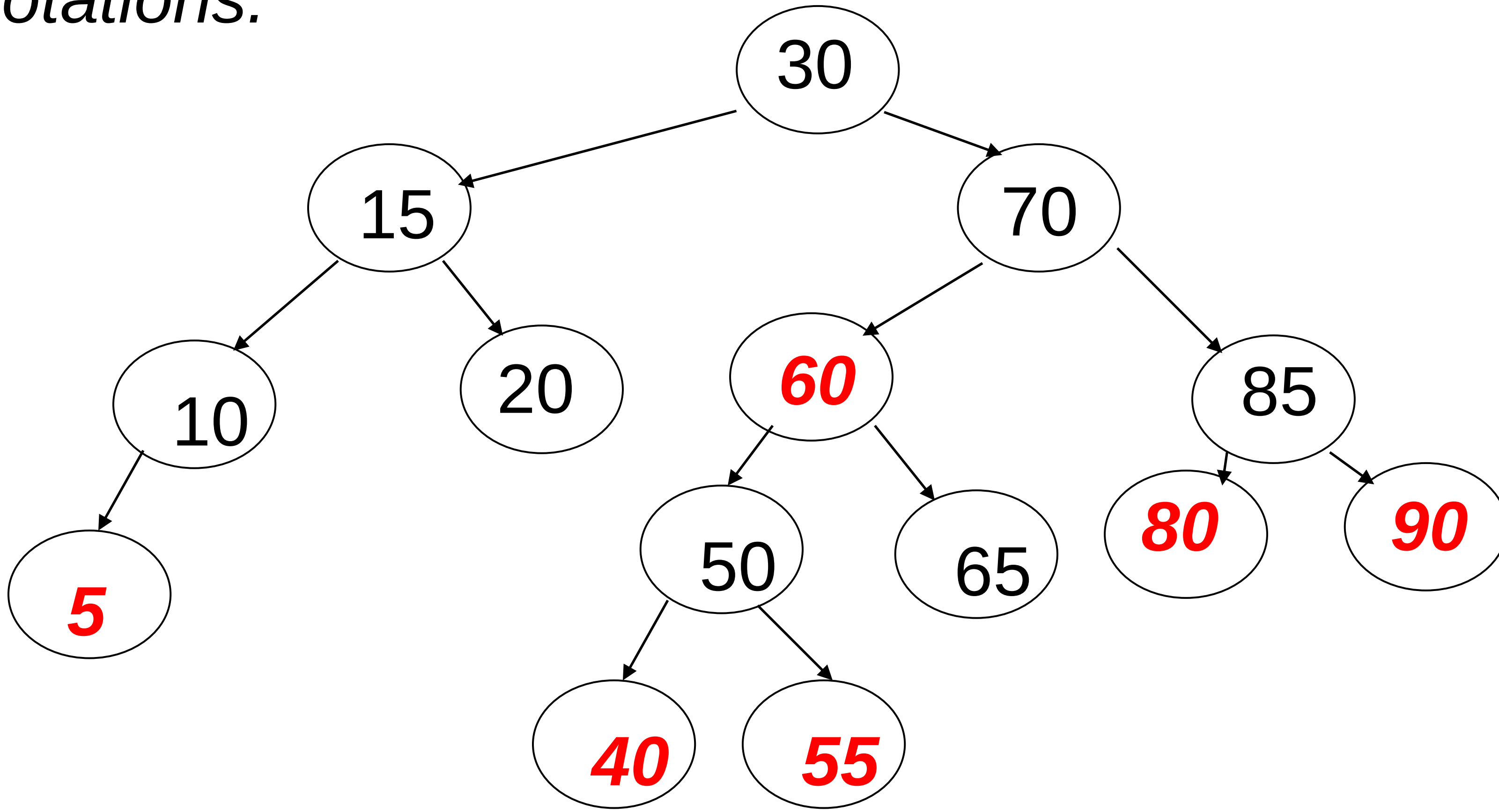
**Recolor: Swap color of S and S's left child**

# Analysis of deletion

- **A red-black tree has O(log *n*) height**
- **Search for deletion location takes O(log *n*) time**
- **The swaping and deletion is O(1).**
- **Each rotation or recoloring is O(1).**
- **Thus, the deletion in a red-black tree takes O(log *n*) time**

# Insertions with Red Parent - Child

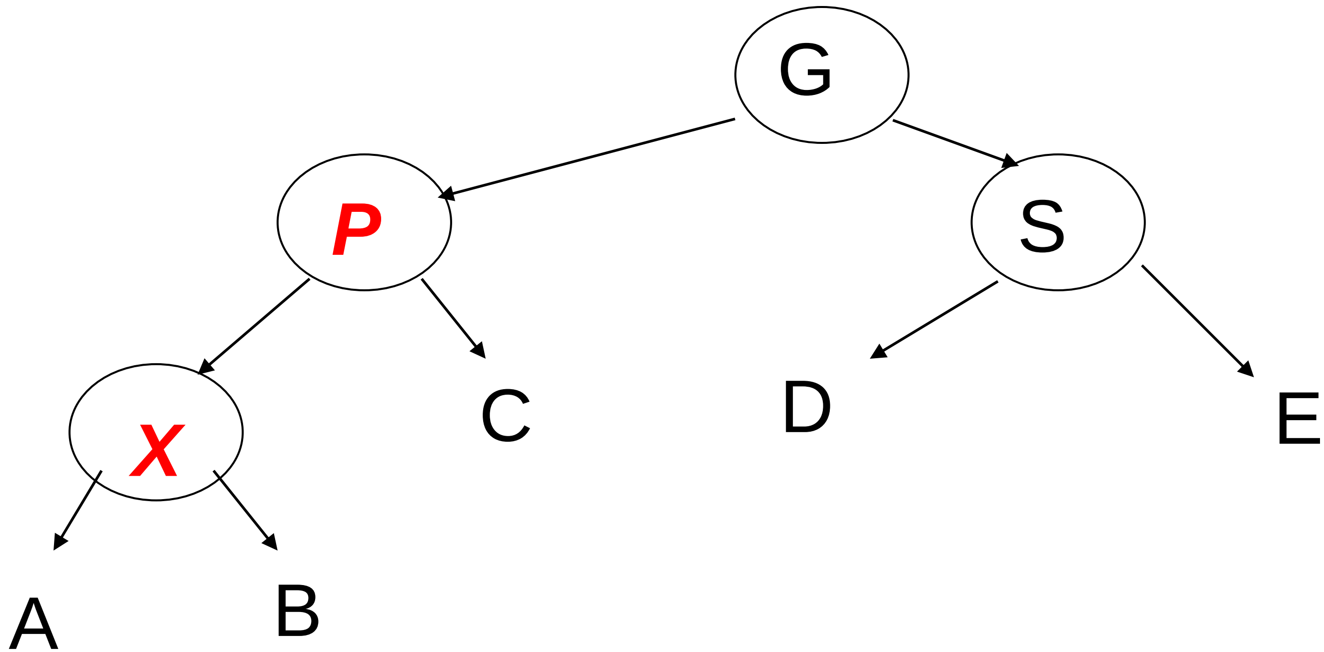
Must modify tree when insertion would result in Red Parent - Child pair using color changes and *rotations.*



# Case 1

- Suppose sibling of parent is Black.
  - by convention null nodes are black
- In the previous tree, true if we are inserting a 3 or an 8.
  - What about inserting a 99? Same case?
- Let X be the new leaf Node, P be its Red Parent, S the Black sibling and G, P's and S's parent and X's grandparent
  - What color is G?

# Case 1 - The Picture

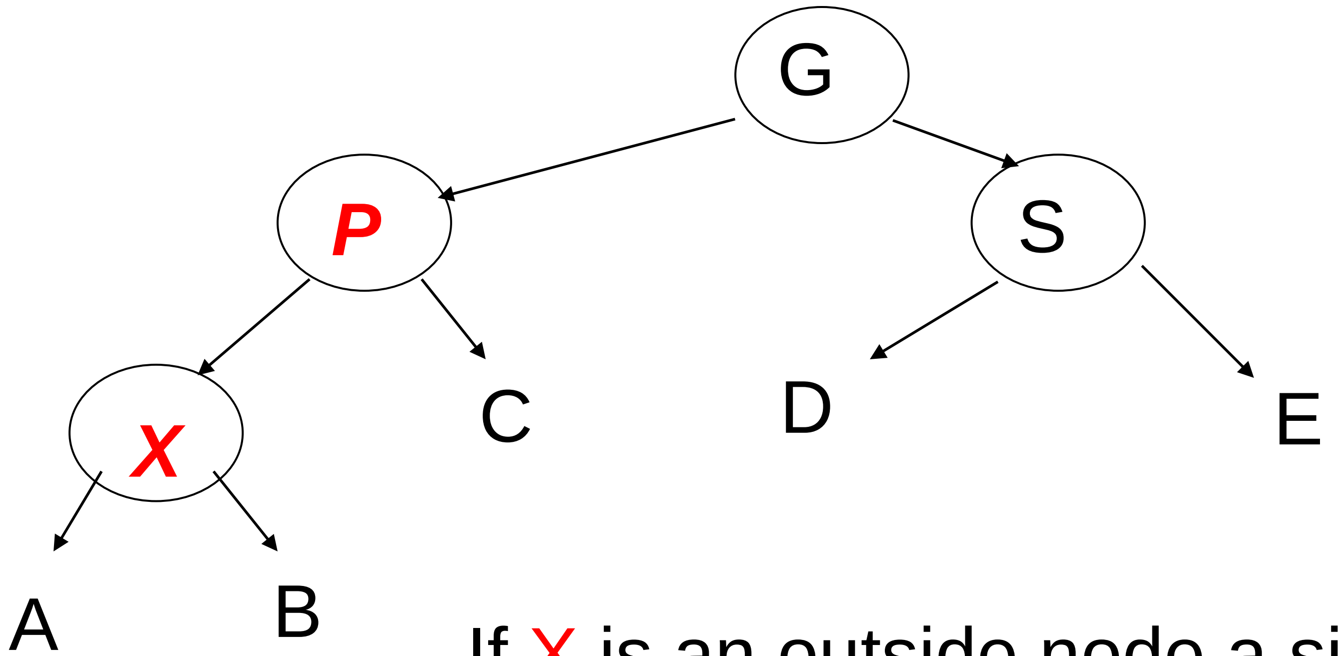



Relative to G, X could be an *inside* or *outside* node.

Outside -> left left or right right moves

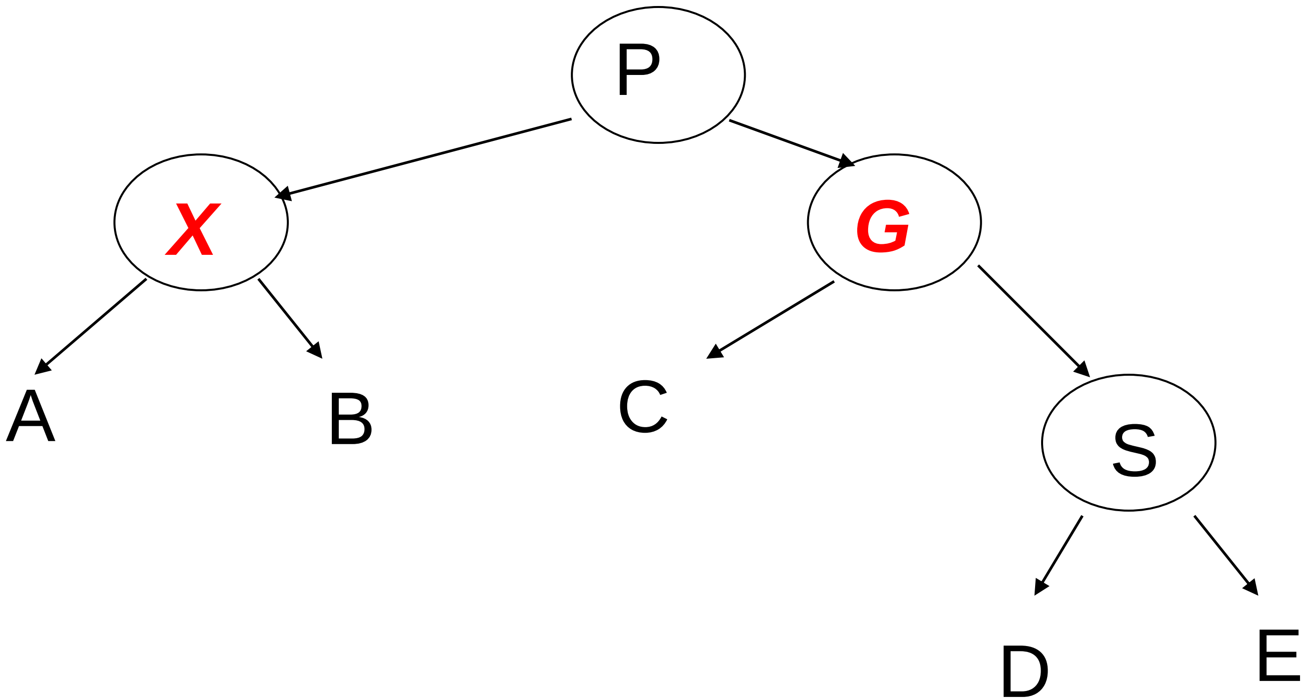
Inside -> left right or right left moves

# Fixing the Problem




If X is an outside node a single *rotation* between P and G fixes the problem.
A rotation is an exchange of roles between a parent and child node. So P becomes G's parent. Also must recolor P and G.

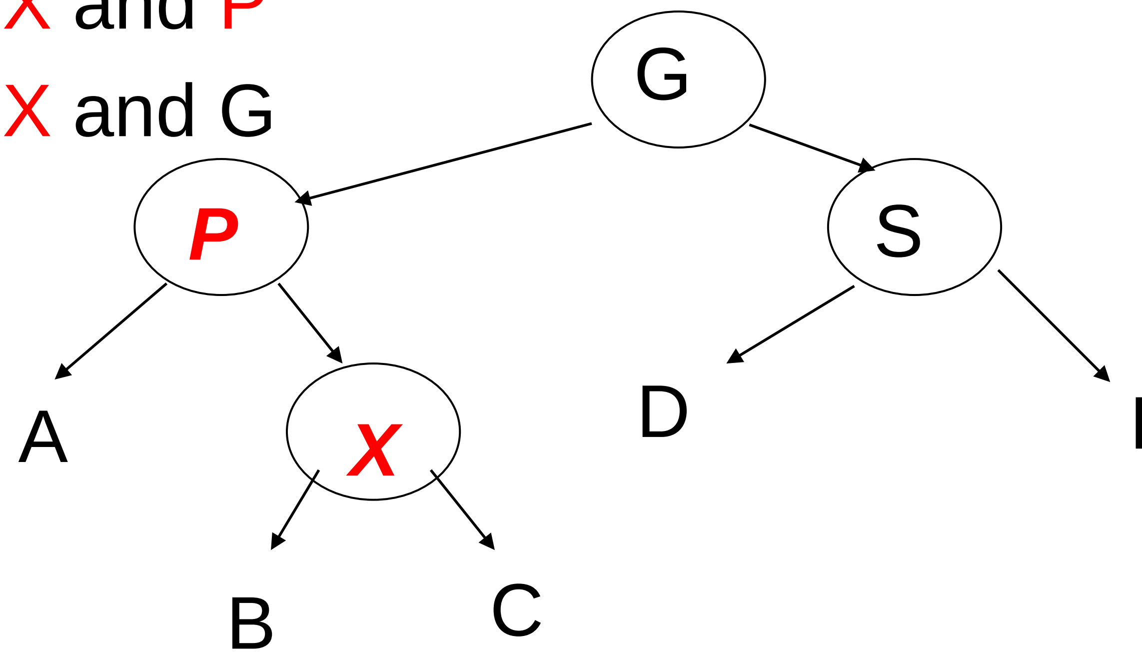
# Single Rotation



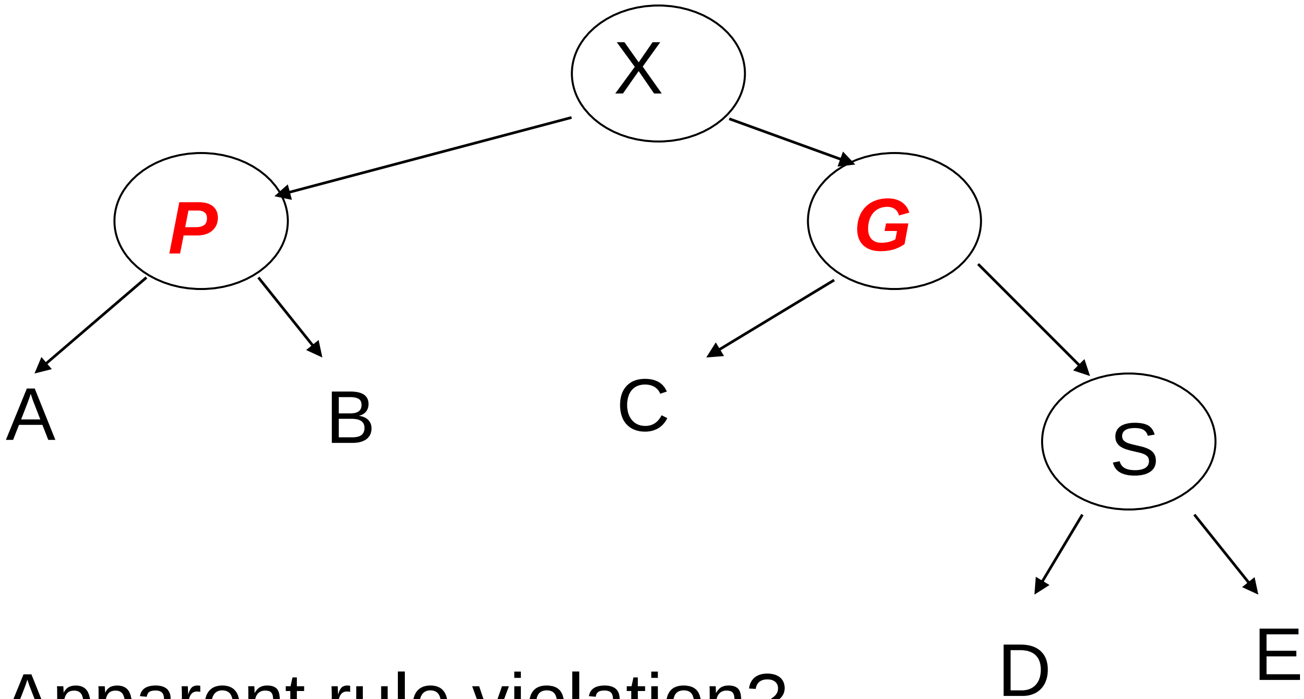

Apparent rule violation?

# Case 2

- What if X is an inside node relative to G?
  – a single rotation will not work
- Must perform a double rotation
  – rotate X and P
  – rotate X and G

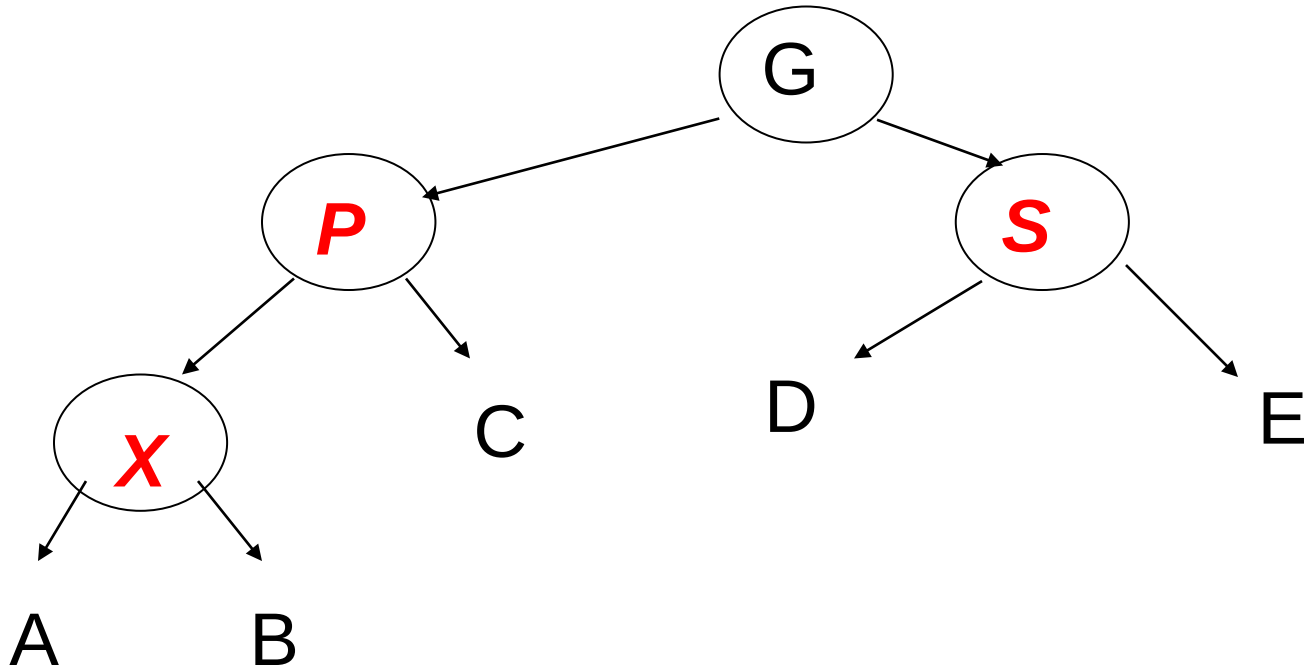


# After Double Rotation




Apparent rule violation?

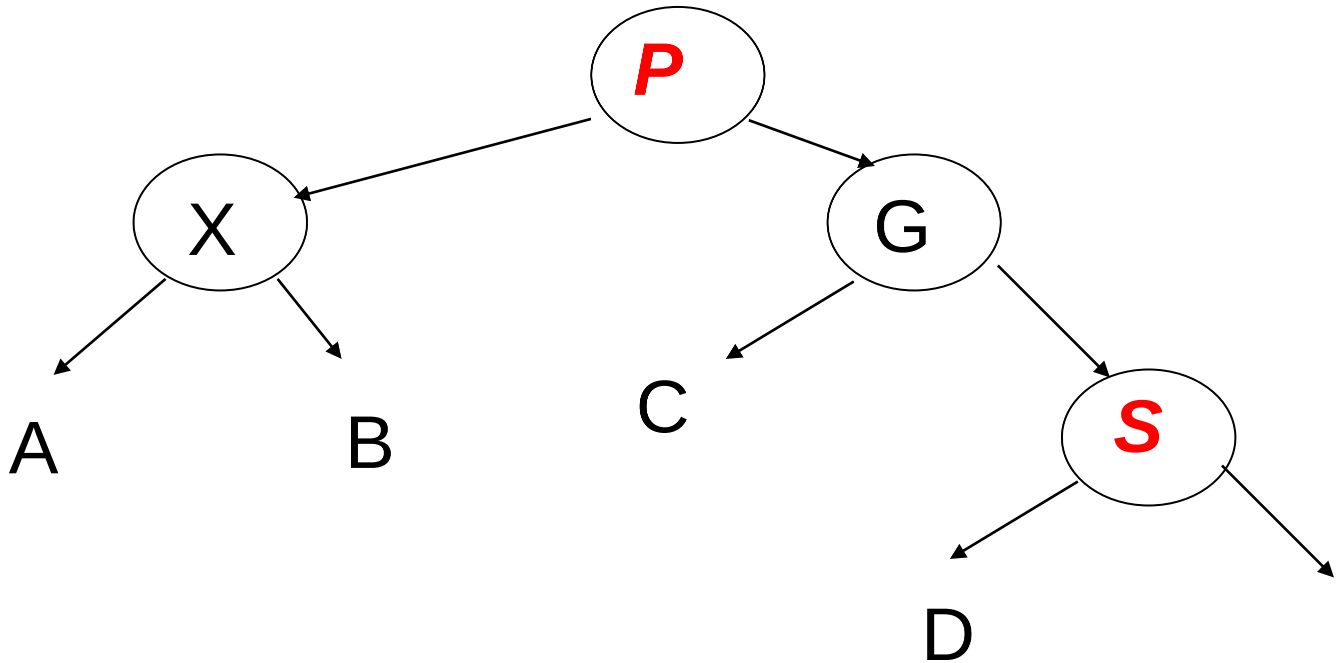
# Case 3
# Sibling is Red, not Black




Any problems?

# Fixing Tree when S is Red

- Must perform single rotation between parent, P and grandparent, G, and then make appropriate color changes

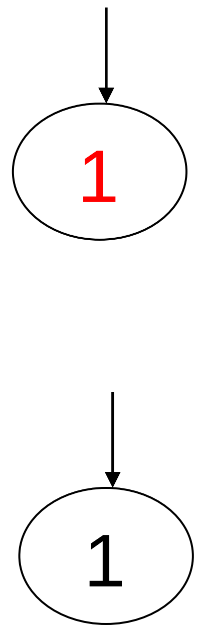


# More on Insert

- Problem: What if on the previous example G's parent had been red?
- Easier to never let Case 3 ever occur!
- On the way down the tree, if we see a node X that has 2 Red children, we make X Red and its two children black.
  - if recolor the root, recolor it to black
  - the number of black nodes on paths below X remains unchanged
  - If X's parent was Red then we have introduced 2 consecutive Red nodes.(violation of rule)
  - to fix, apply rotations to the tree, same as inserting node

# Example of Inserting Sorted Numbers

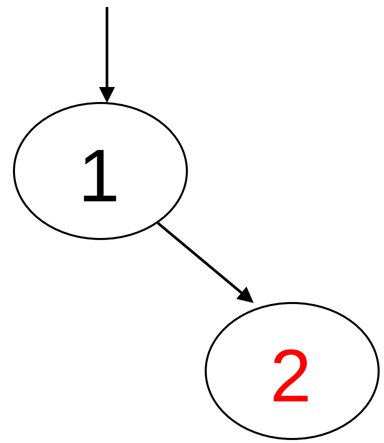
- 1 2 3 4 5 6 7 8 9 10




Insert 1. A leaf so red. Realize it is root so recolor to black.

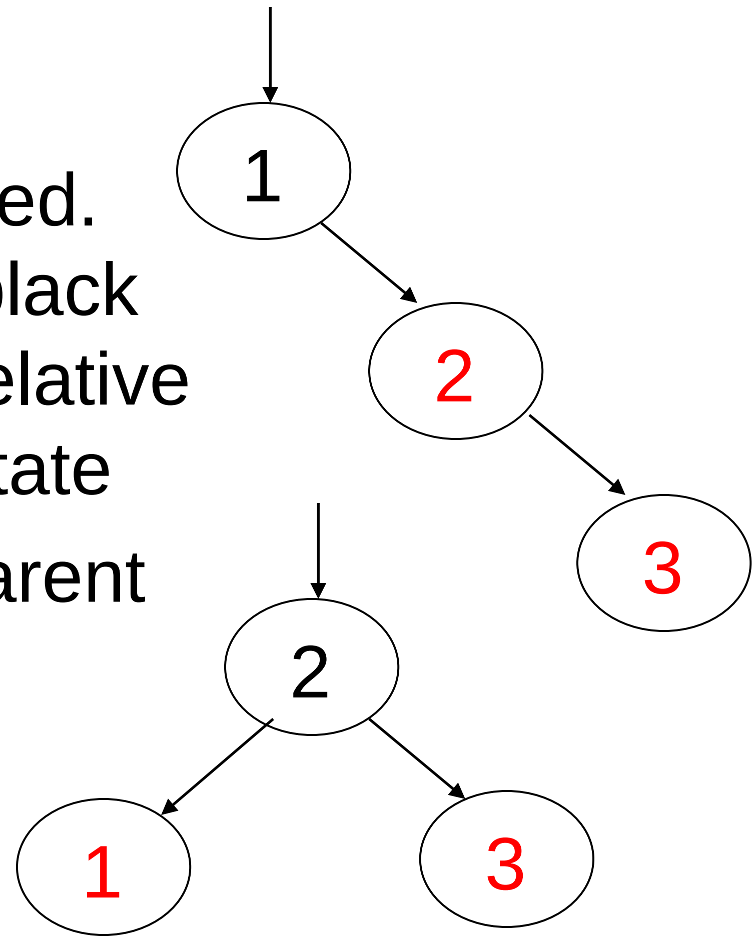
# Insert 2

make 2 red. Parent is black so done.



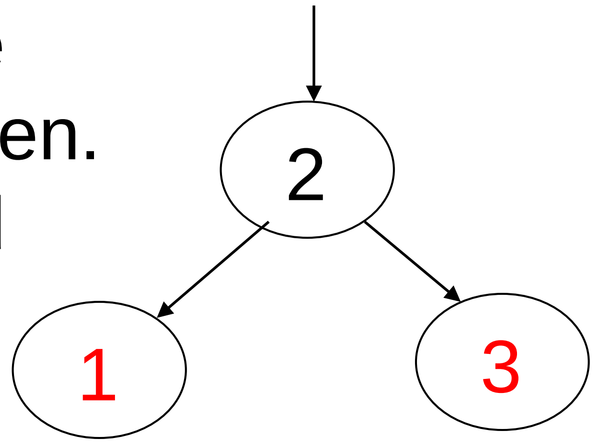
# Insert 3

Insert 3. Parent is red. Parent's sibling is black (null) 3 is outside relative to grandparent. Rotate parent and grandparent

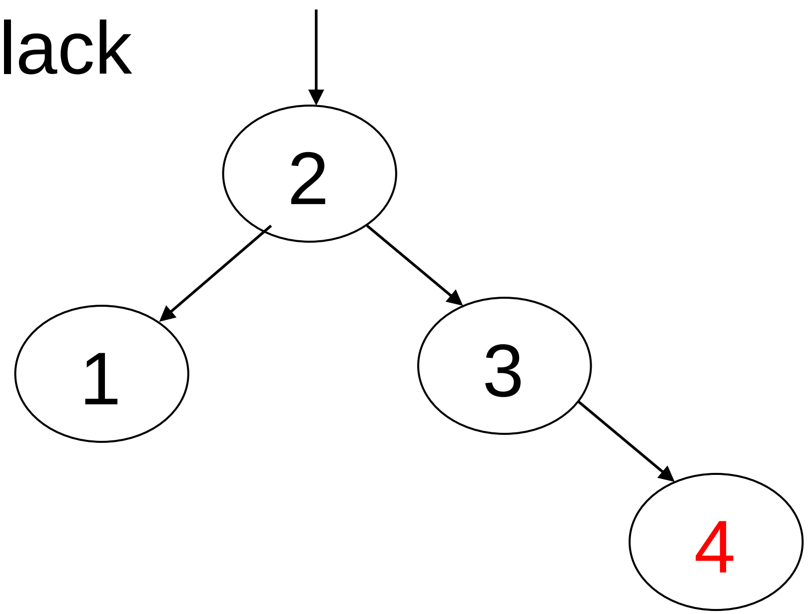


# Insert 4

On way down see
2 with 2 red children.
Recolor 2 red and
children black.
Realize 2 is root
so color back to black

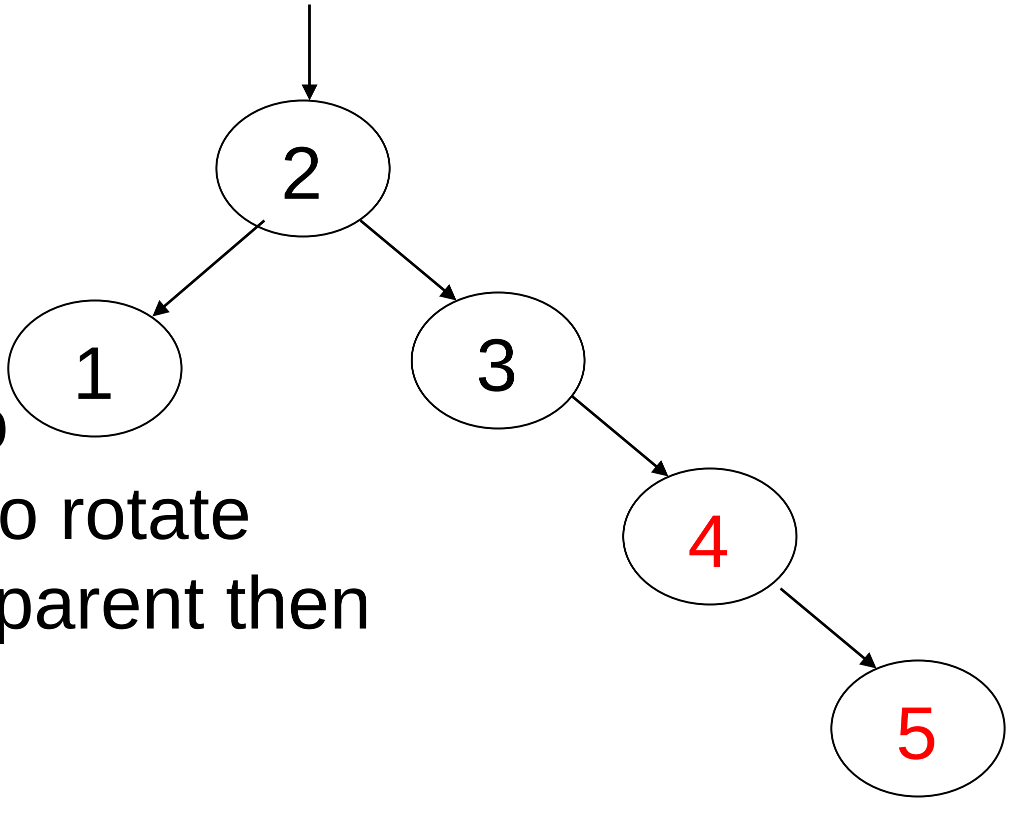



When adding 4
parent is black
so done.

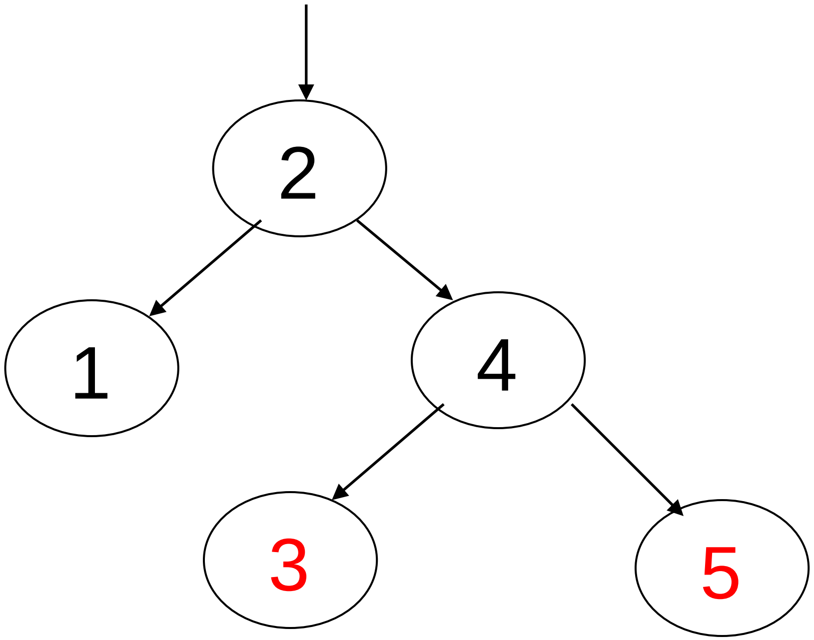


# Insert 5

5's parent is red. Parent's sibling is black (null). 5 is outside relative to grandparent (3) so rotate parent and grandparent then recolor

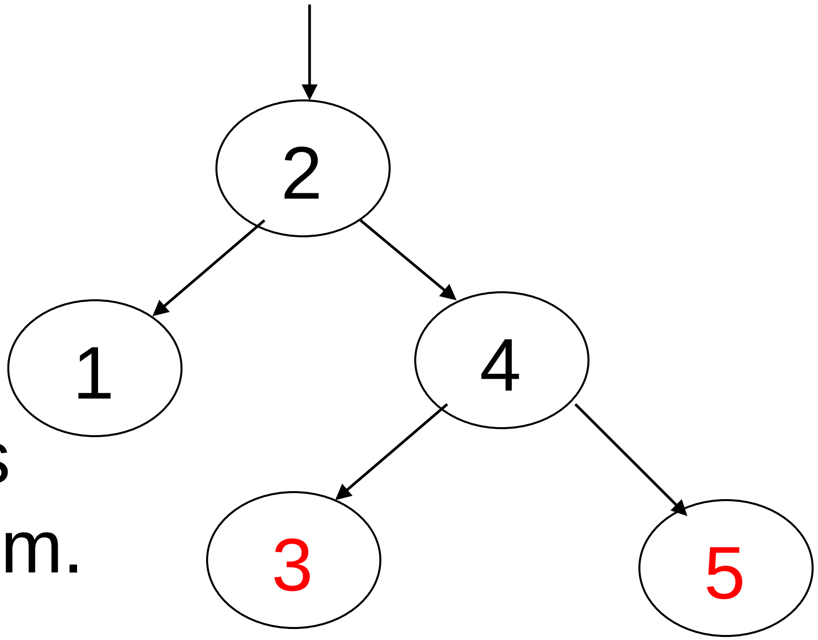


# Finish insert of 5



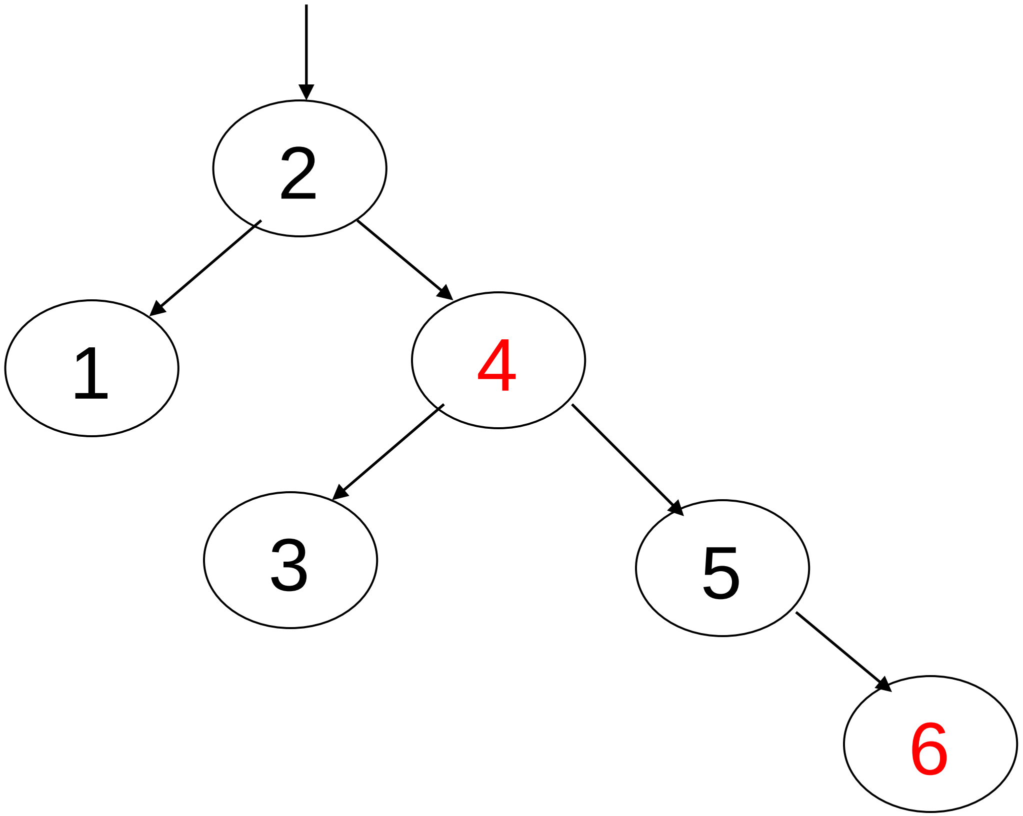
# Insert 6

On way down see 4 with 2 red children. Make 4 red and children black. 4's parent is black so no problem.



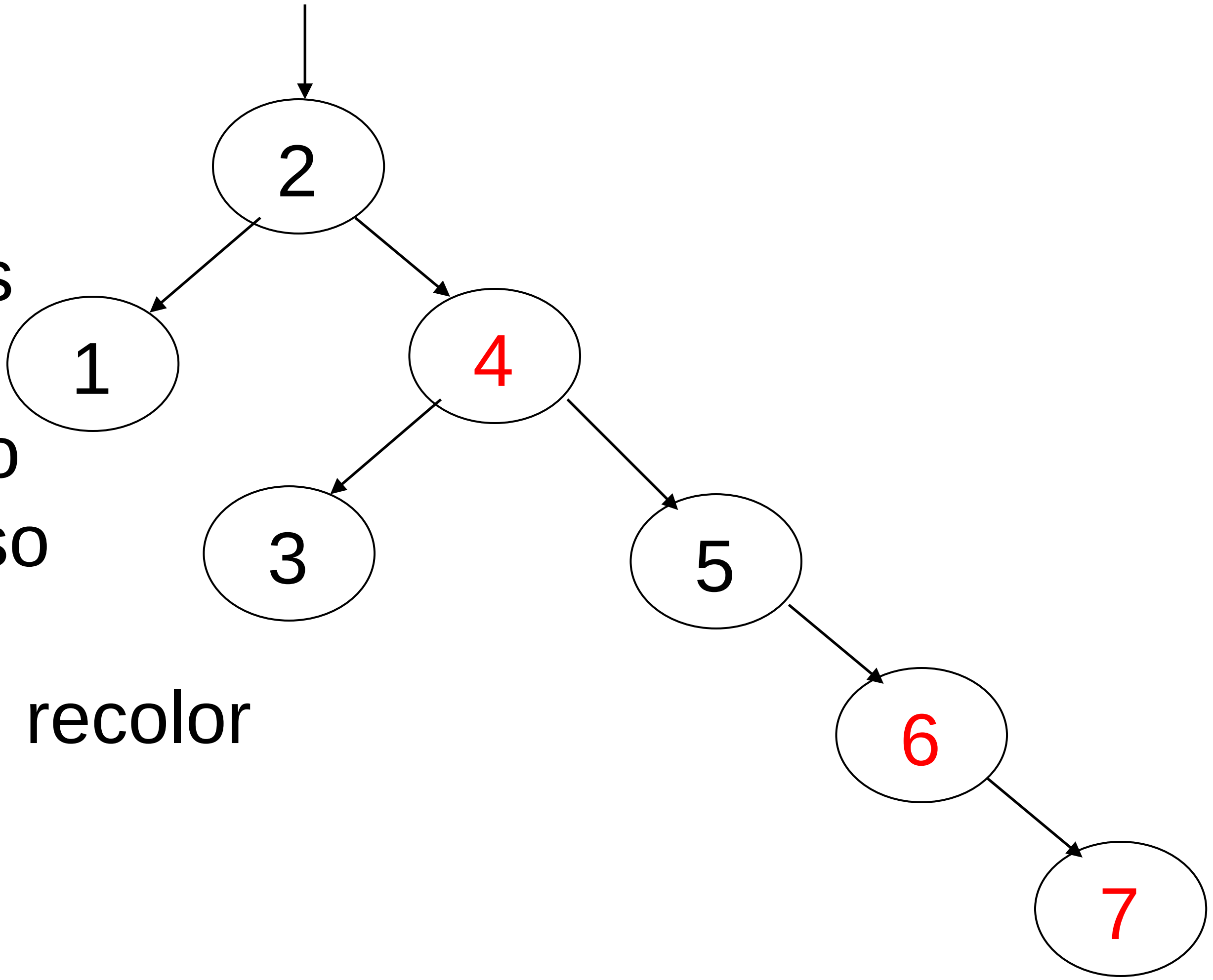
# Finishing insert of 6

6's parent is black so done.

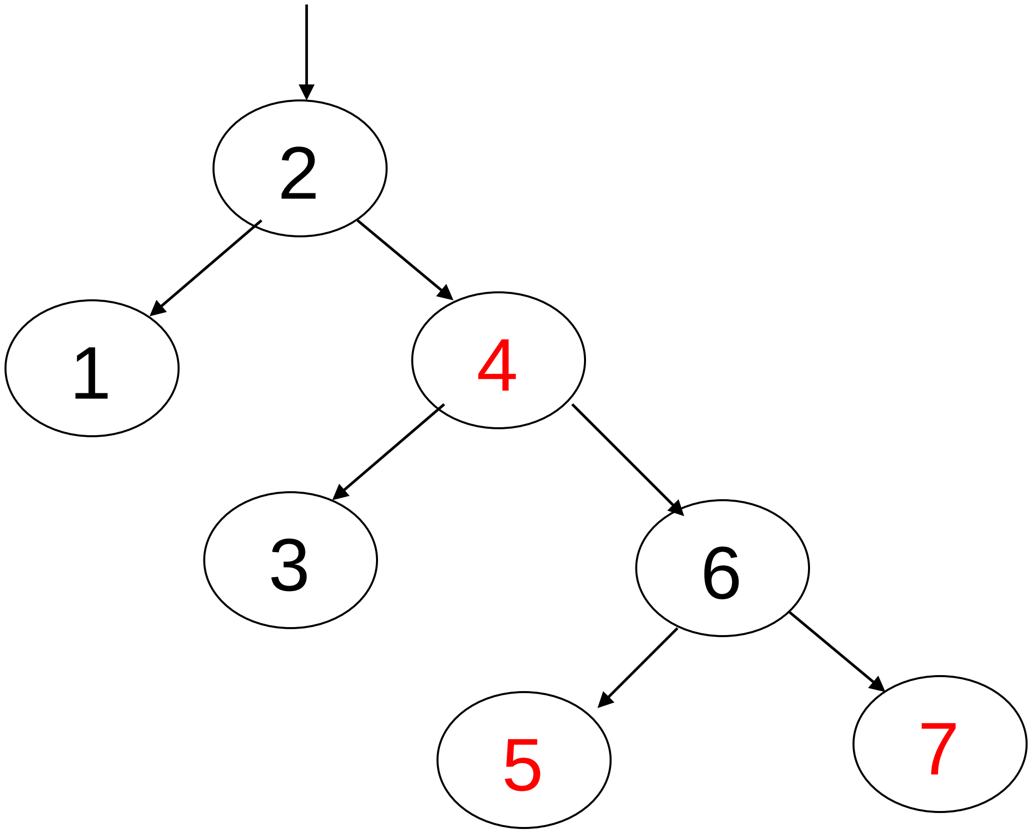


# Insert 7

7's parent is red. Parent's sibling is black (null). 7 is outside relative to grandparent (5) so rotate parent and grandparent then recolor

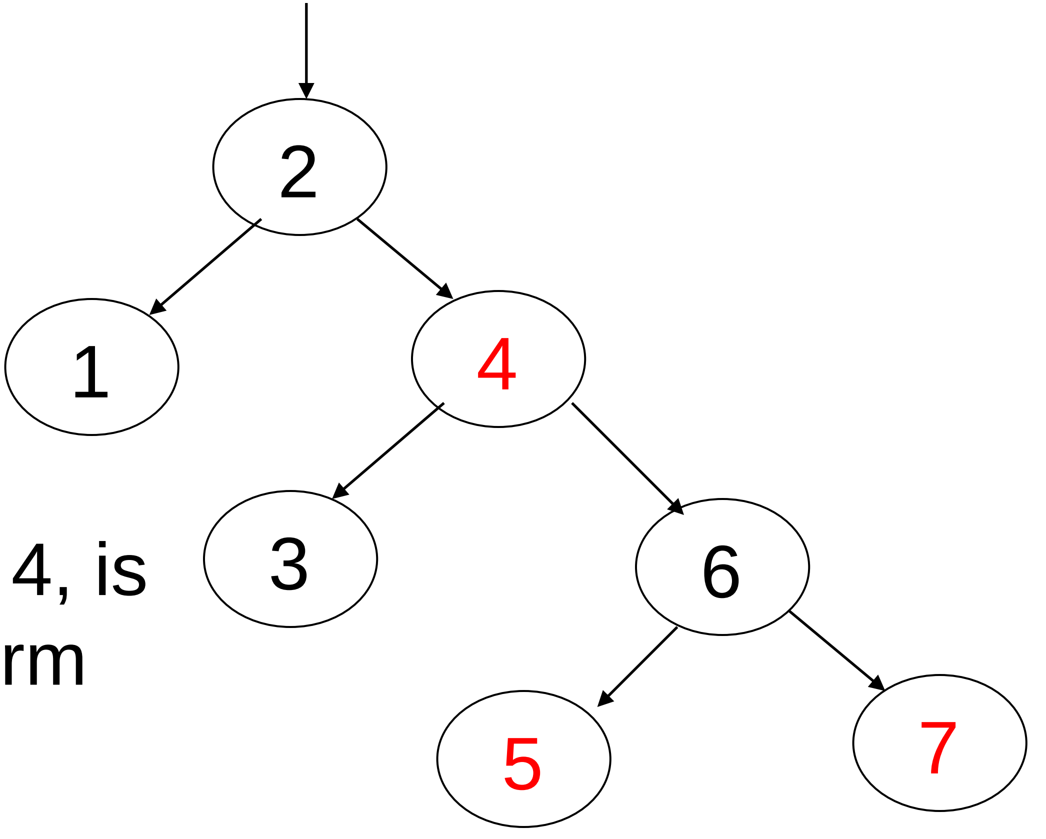


# Finish insert of 7



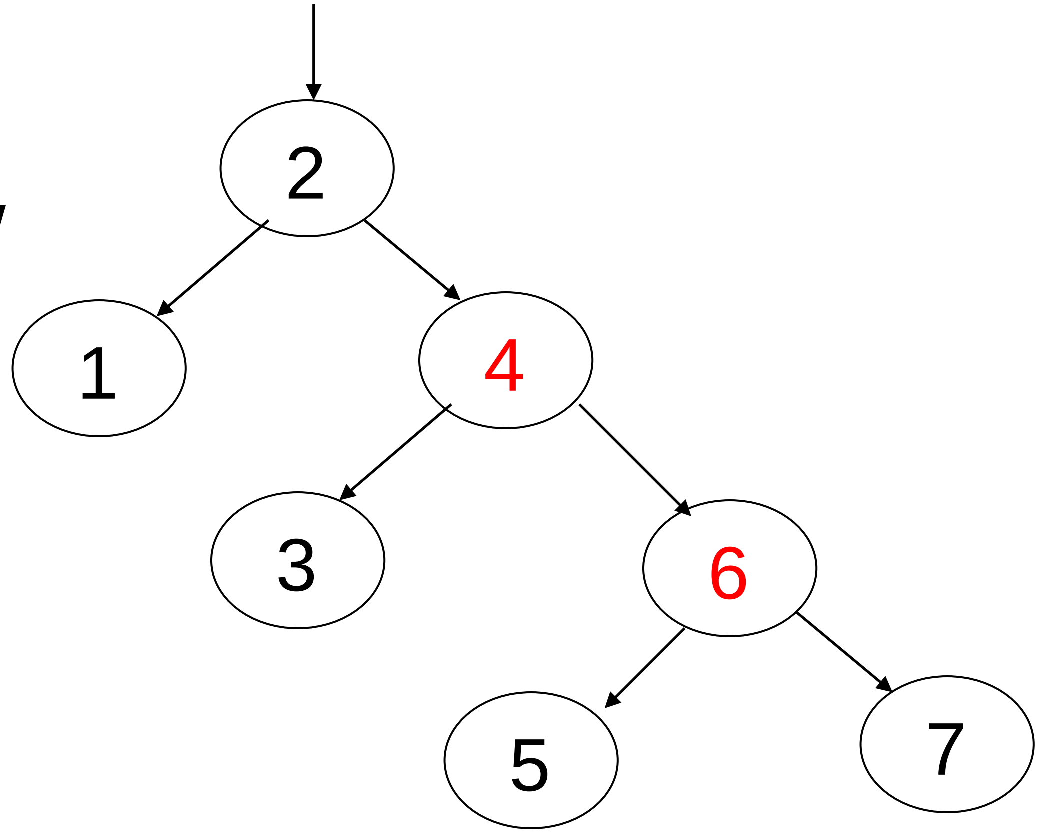
# Insert 8

On way down see 6 with 2 red children. Make 6 red and children black. This creates a problem because 6's parent, 4, is also red. Must perform rotation.



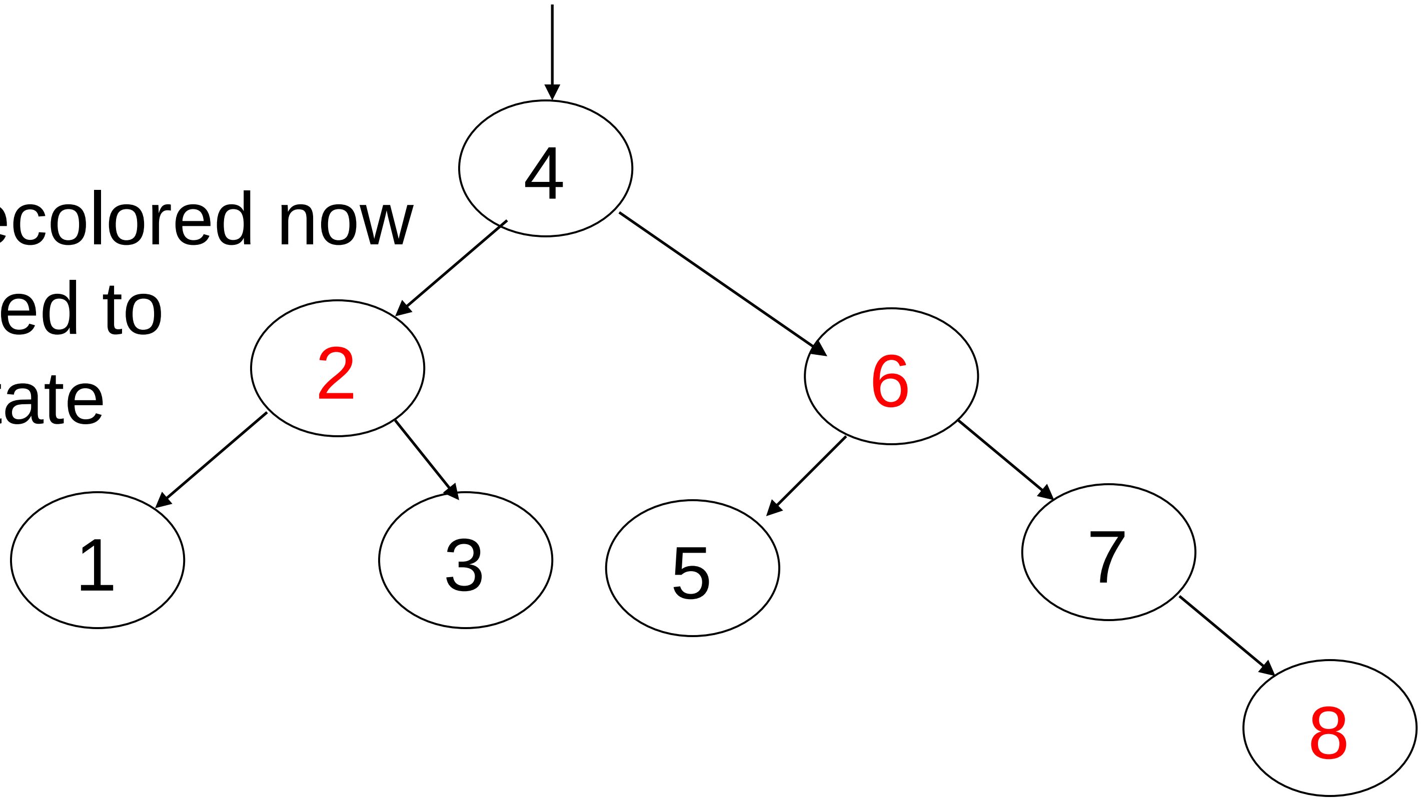
# Still Inserting 8

Recolored now need to rotate

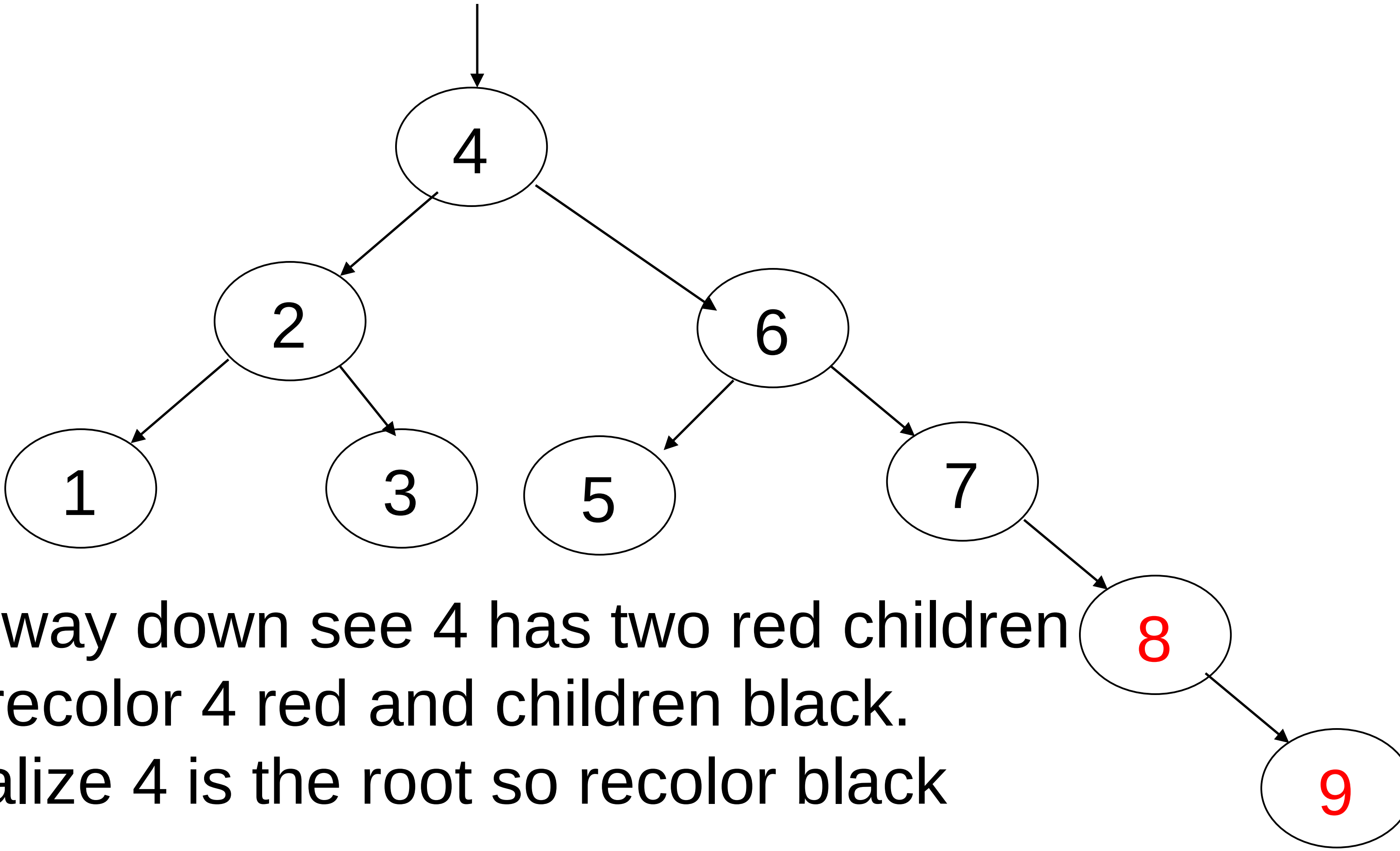


# Finish inserting 8

Recolored now need to rotate

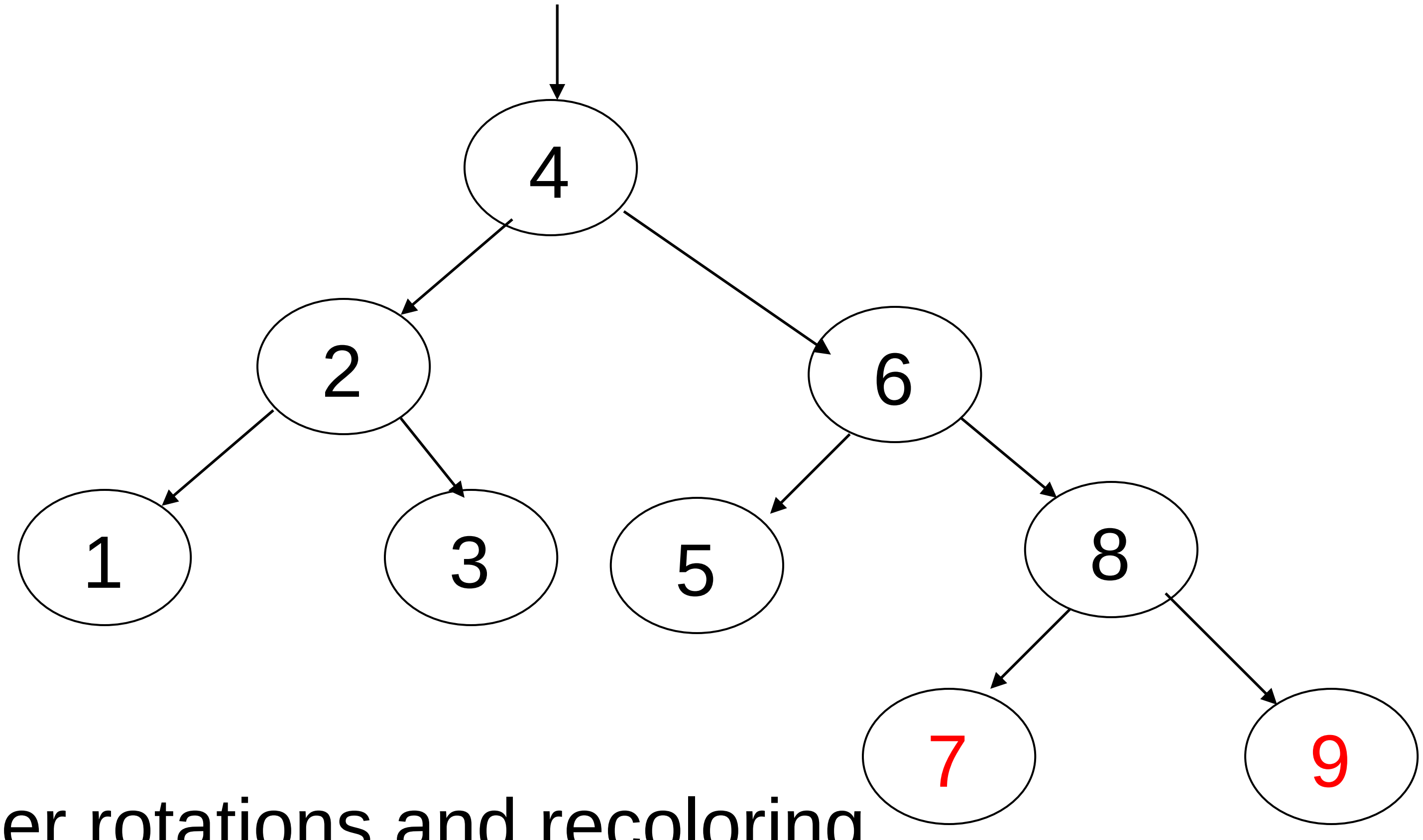


# Insert 9



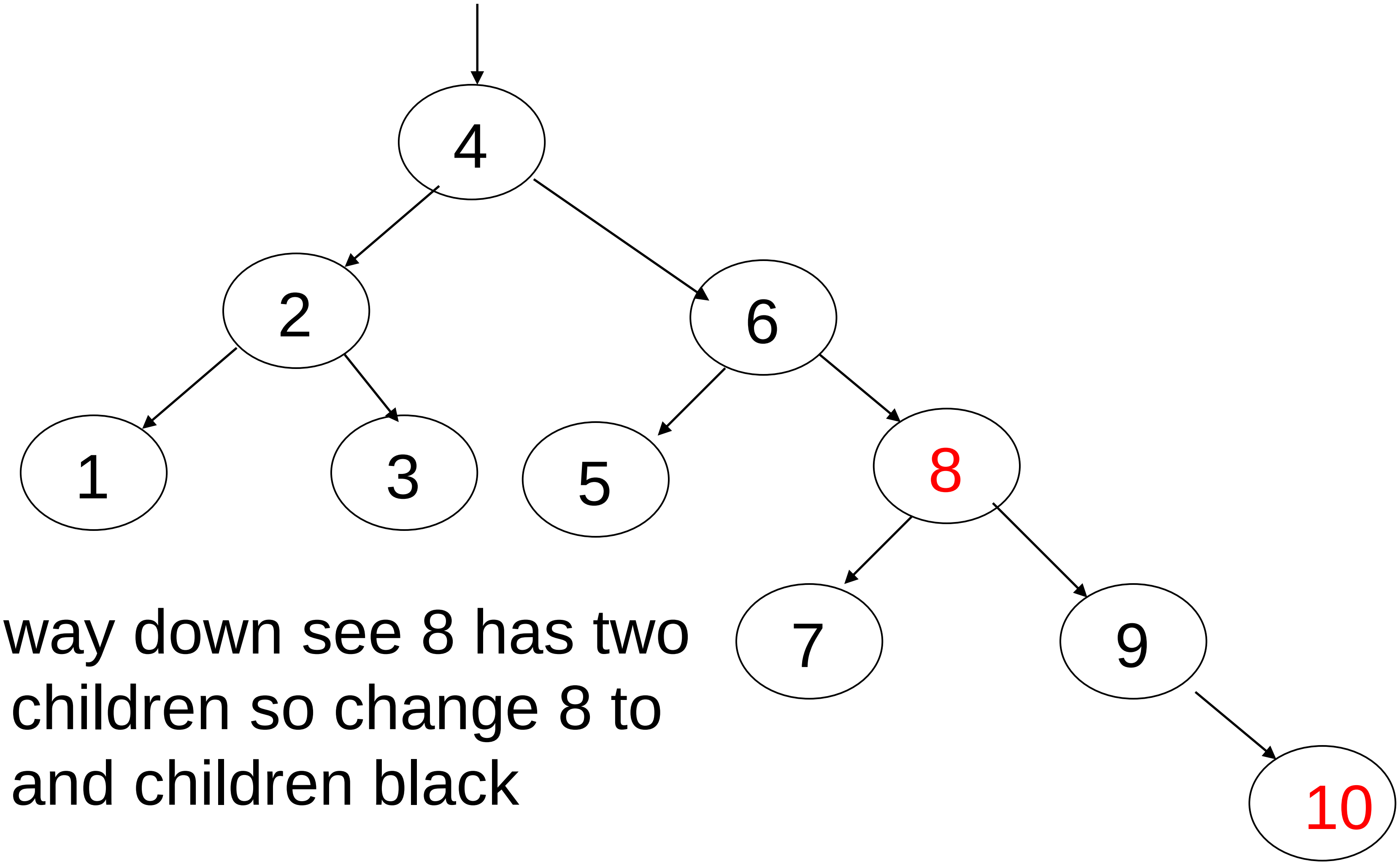

On way down see 4 has two red children
so recolor 4 red and children black.
Realize 4 is the root so recolor black

# Finish Inserting 9



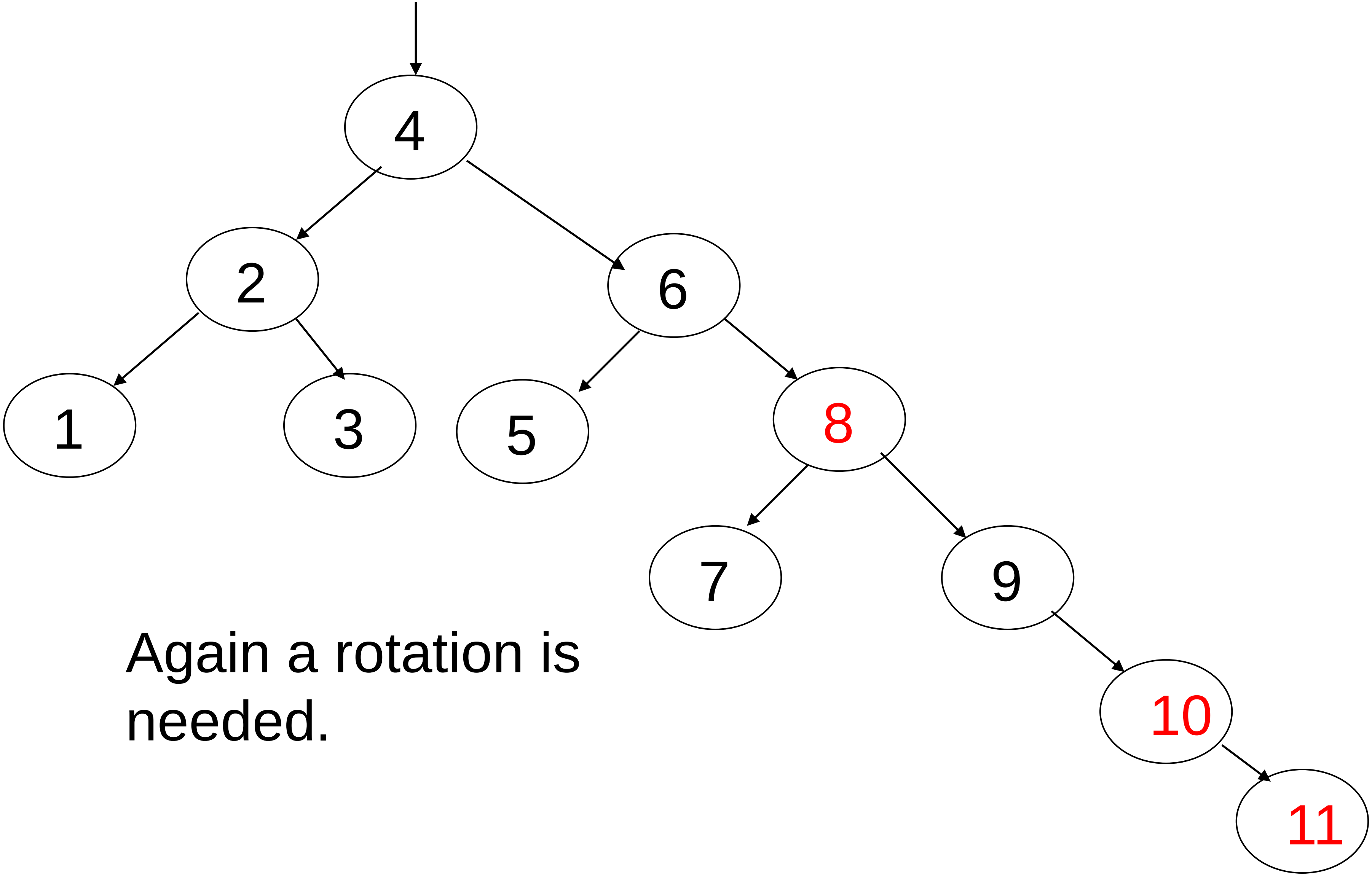

After rotations and recoloring

# Insert 10



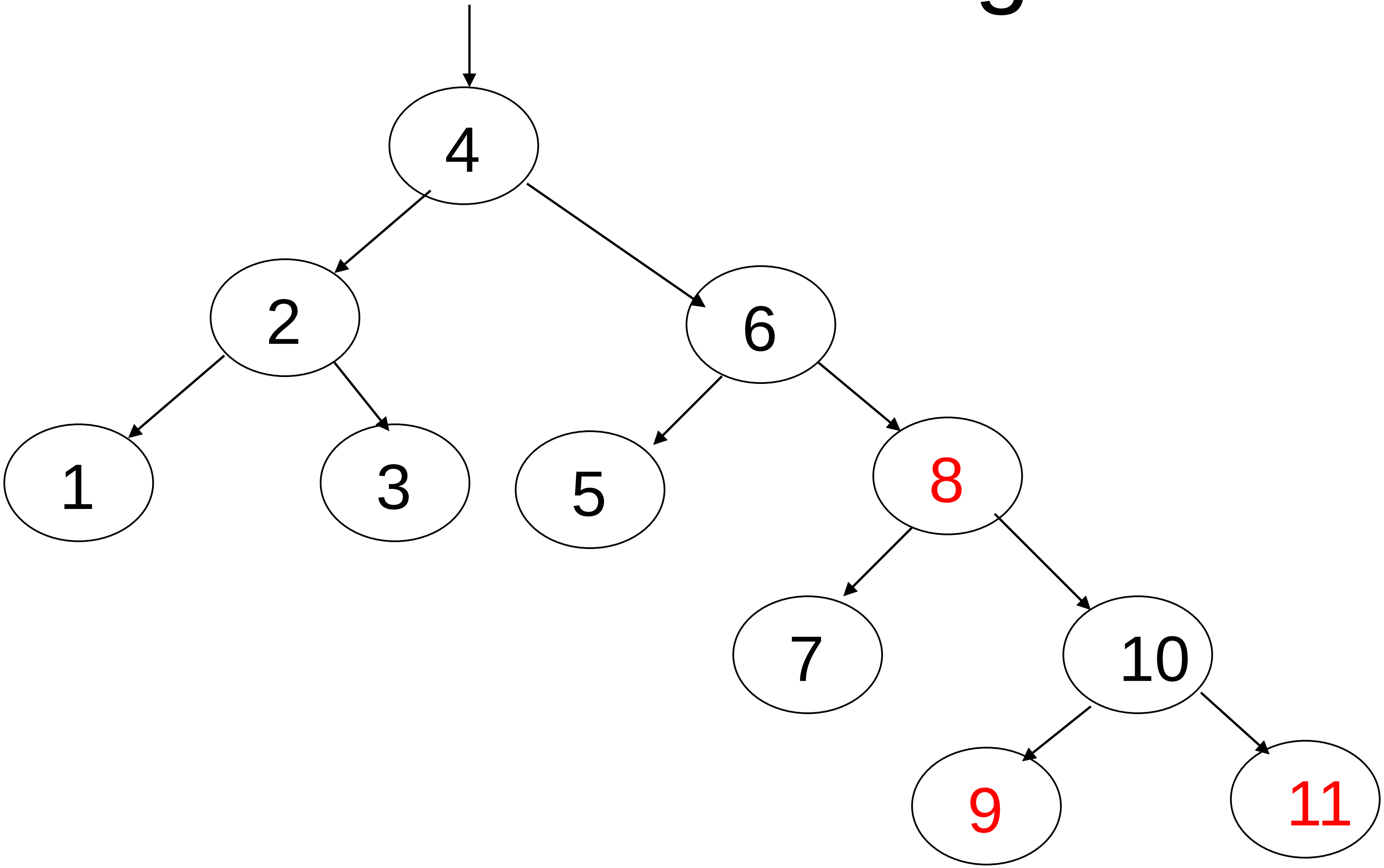

On way down see 8 has two red children so change 8 to red and children black

# Insert 11

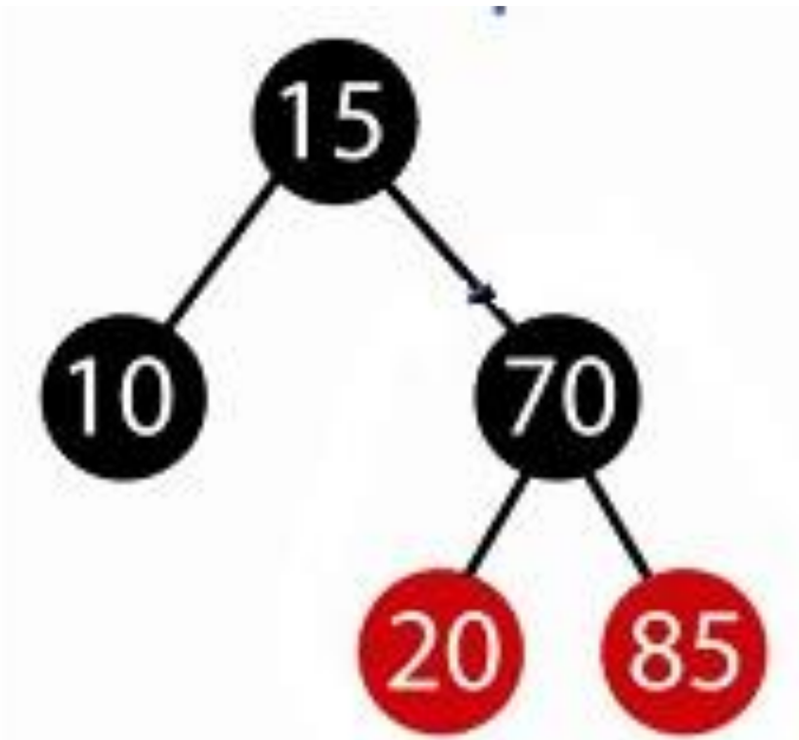


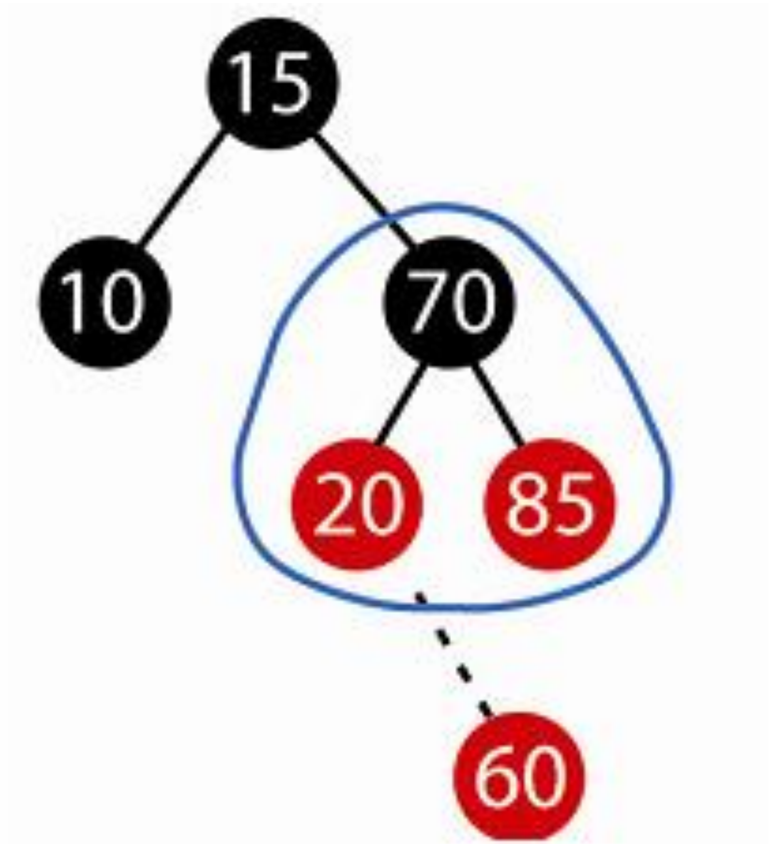

Again a rotation is needed.

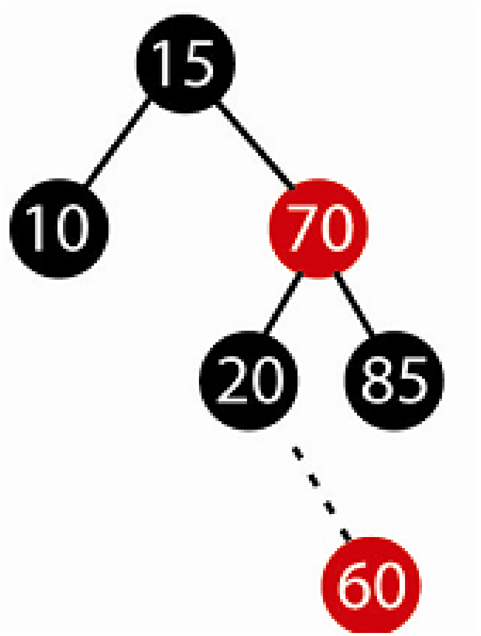
# Finish inserting 11

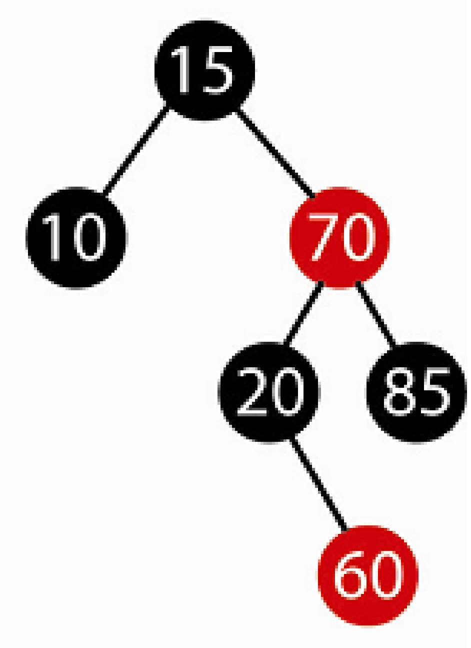


# Other examples

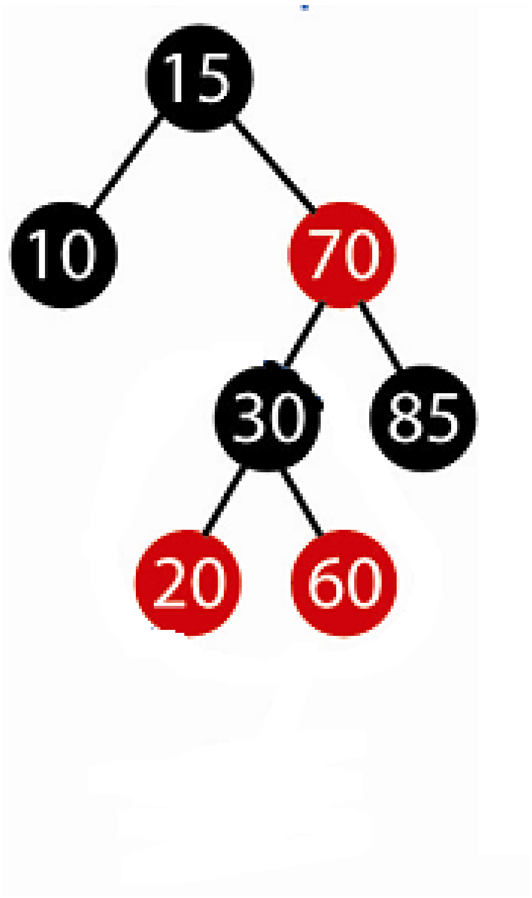

15
10
70
20
85

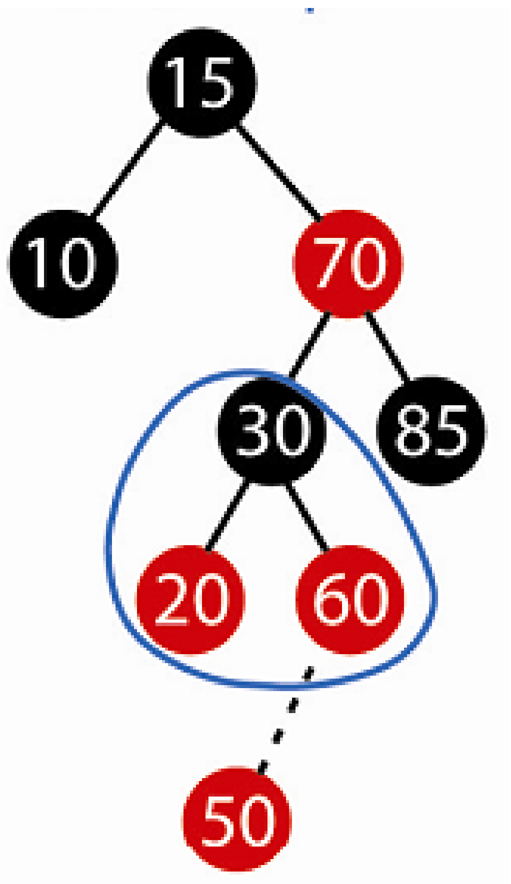

15
10
70
20
85
60

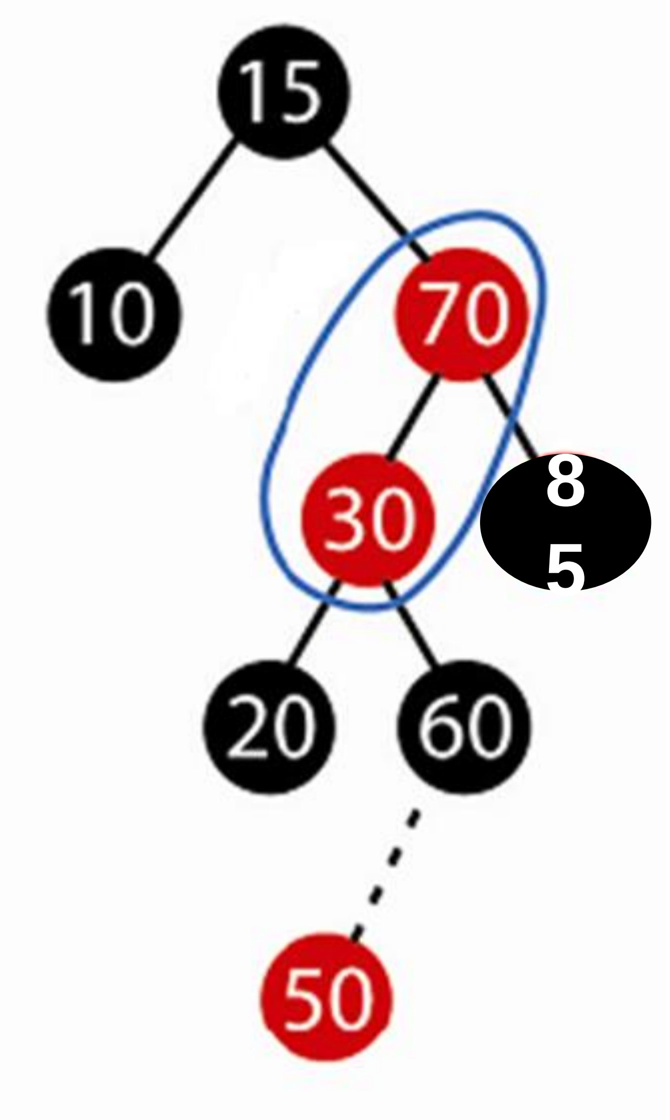

15
10
70
20
85
60

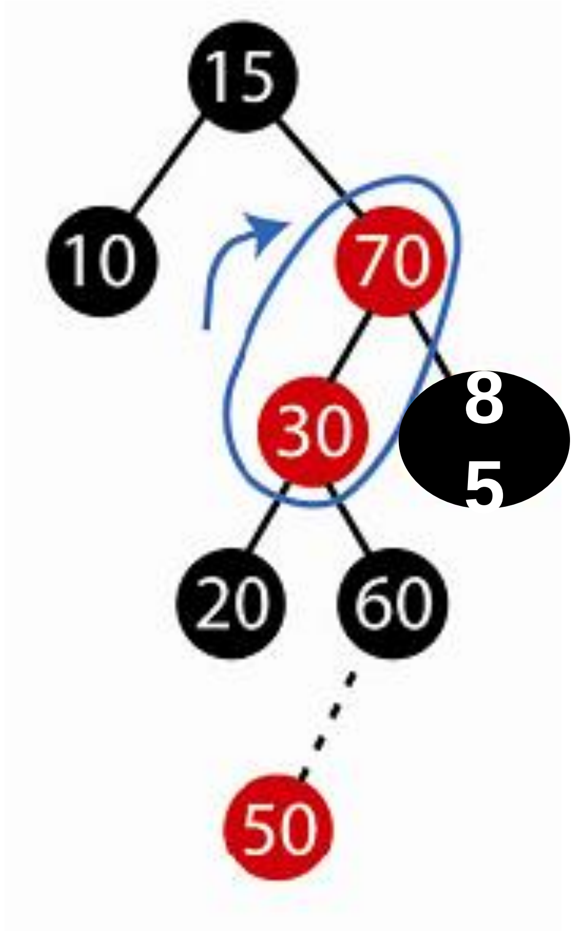

15
10
70
20
85
60

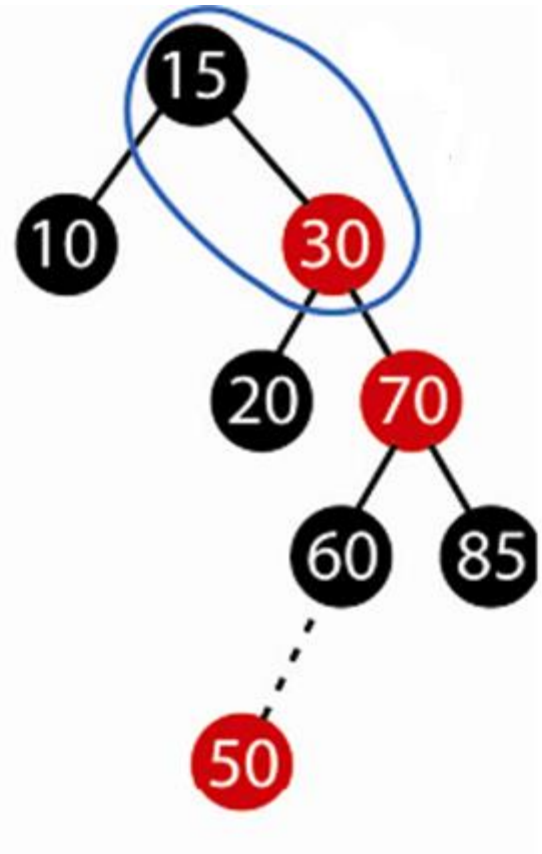
# Another Example

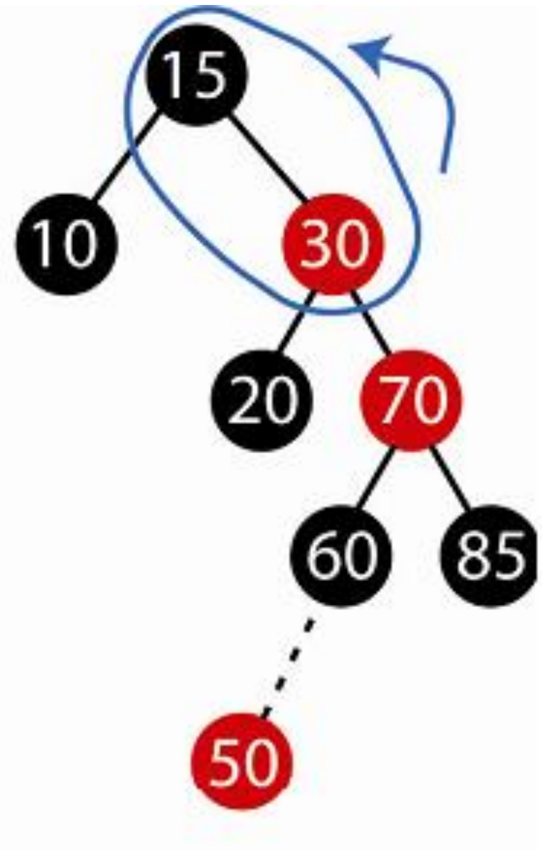


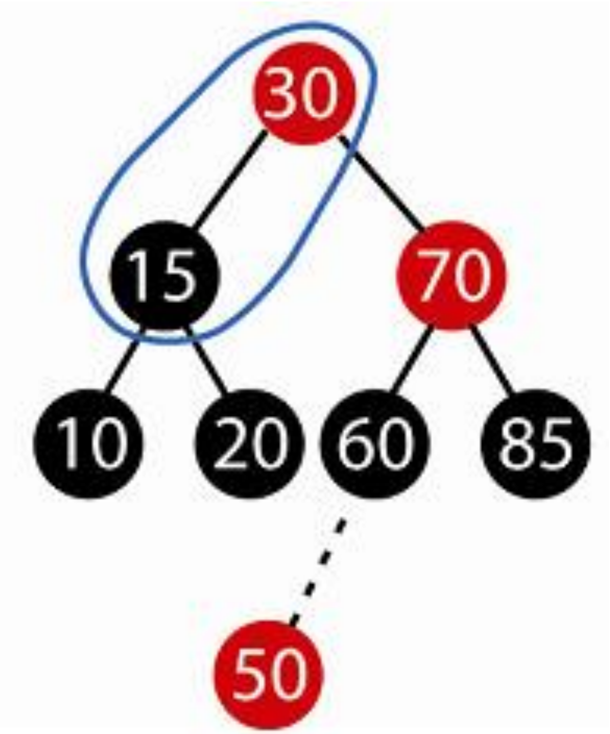

15
10
70
30
85
20
60
50

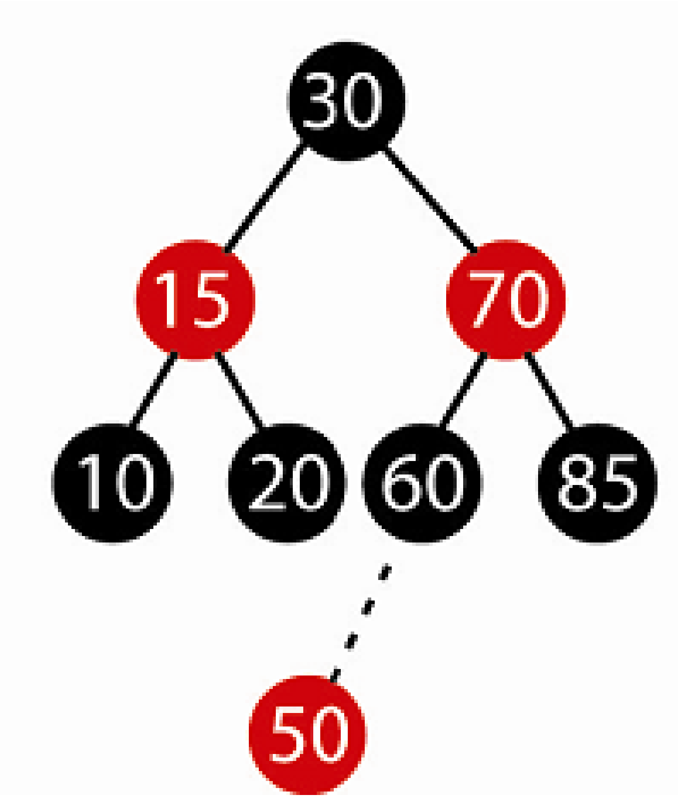

15
10
70
30
8
5
20
60
50

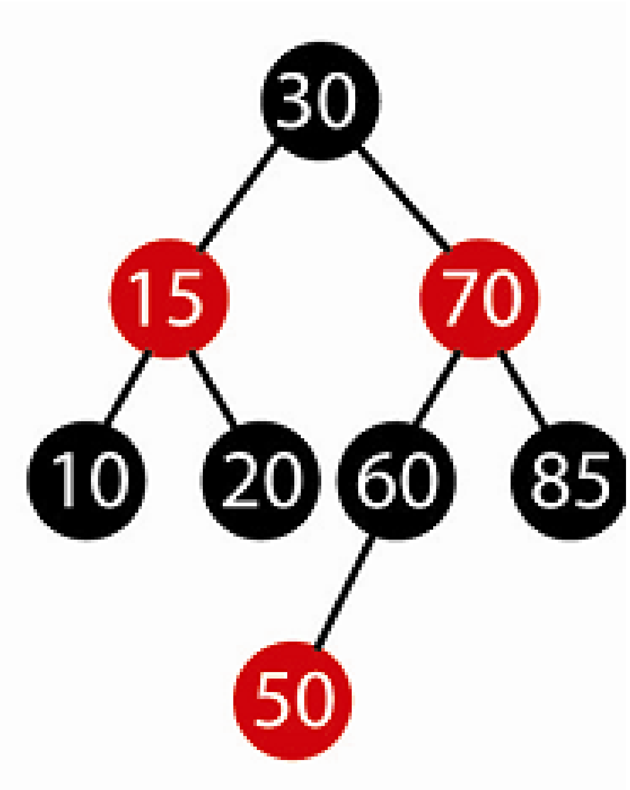

15
10
70
30
8
5
20
60
50


15
10
30
20
70
60
85
50


15
10
30
20
70
60
85
50


30
15
70
10
20
60
85
50


30
15
70
10
20
60
85
50


30
15
70
10
20
60
85
50