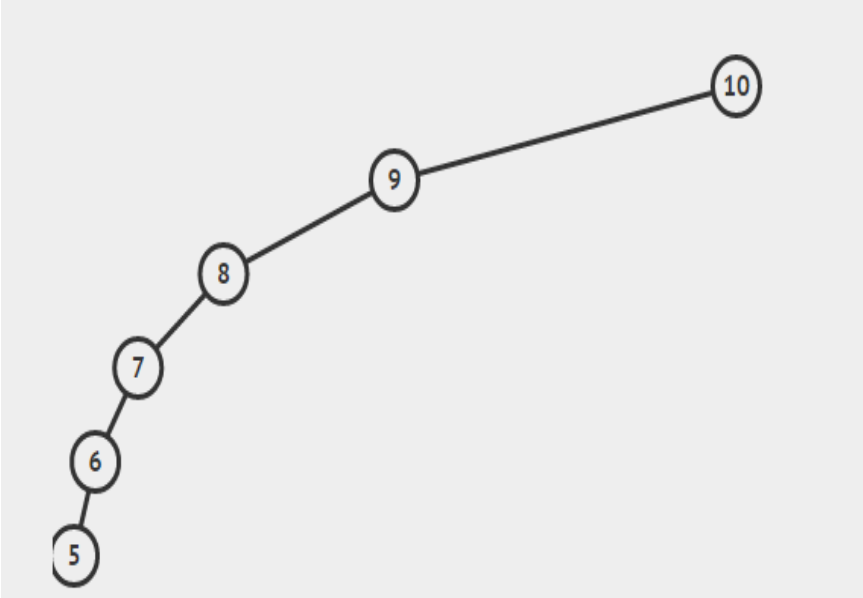
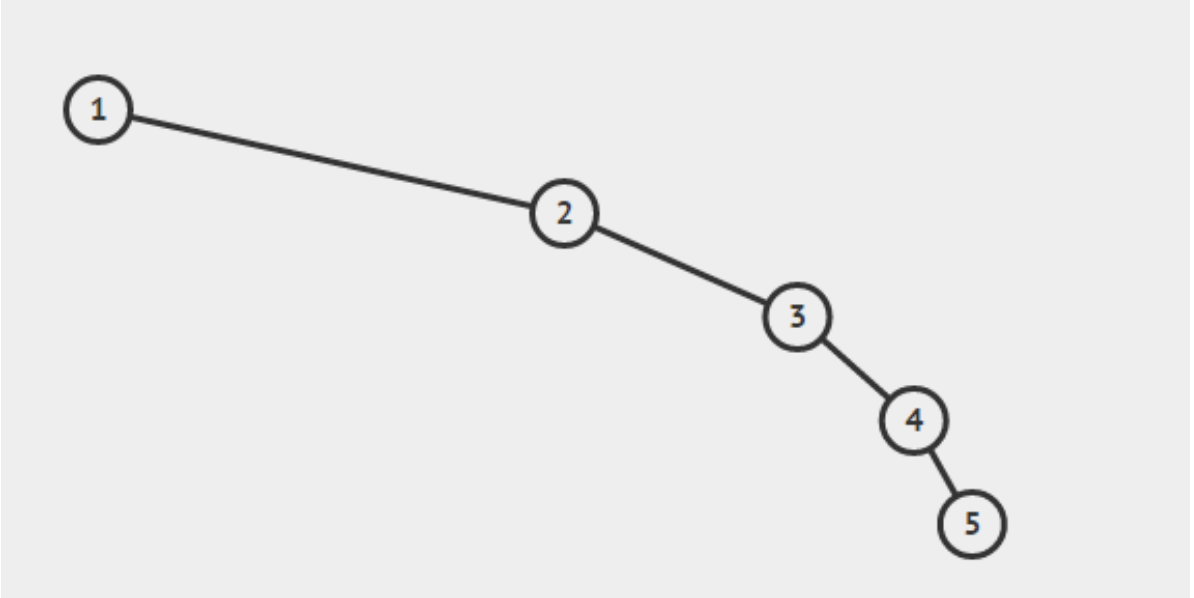
AVL Tree
A Balanced Binary Search Tree

Revised based on textbook author's notes and Professor Booth's notes.

How bad!?!?

- Balance – the height of left and right subtree approximately equal
- Our standard binary trees can be bad!
  - What if we made a search tree from an ordered list?

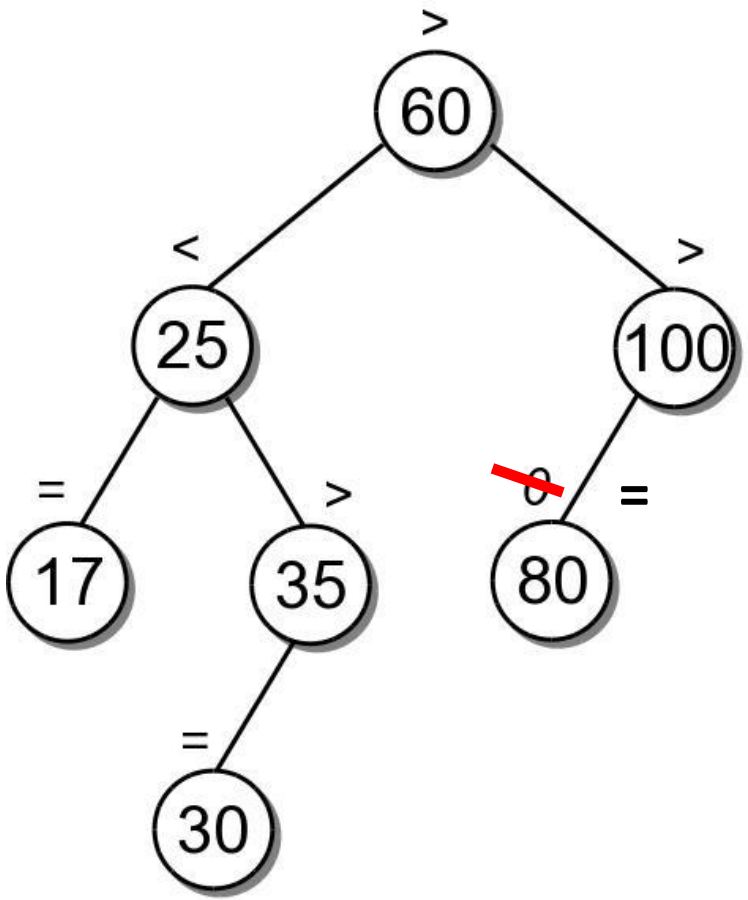


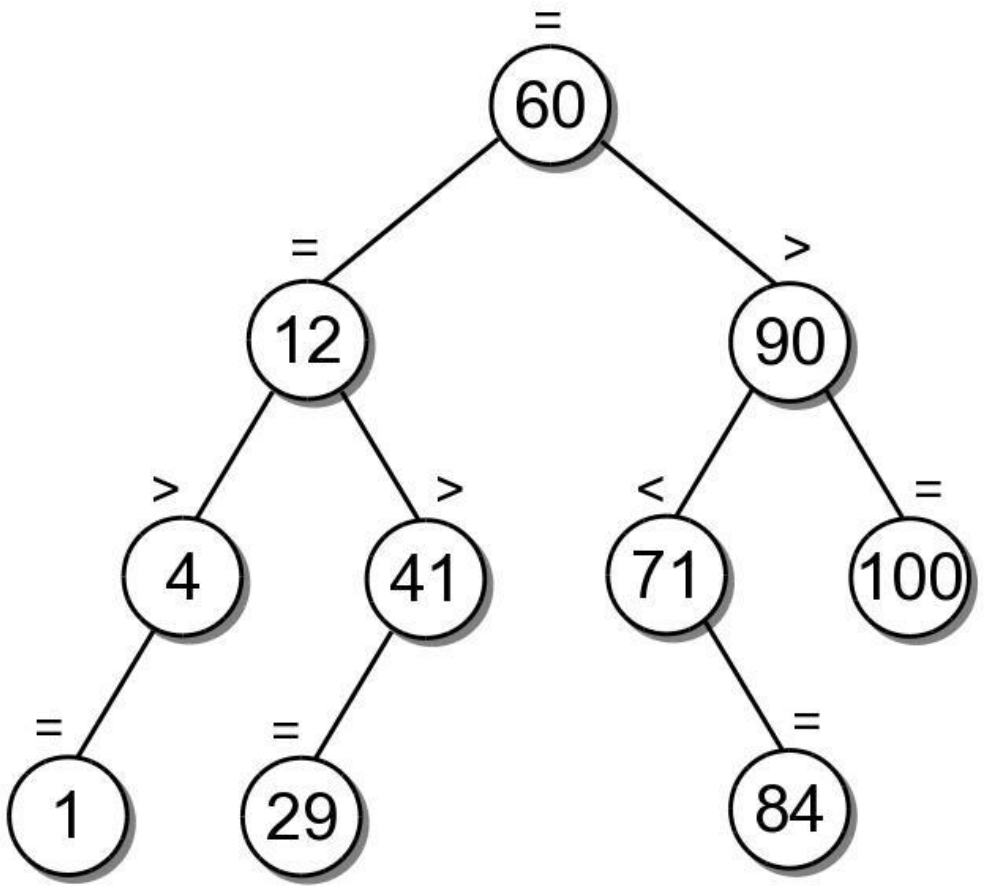


AVL Tree Background

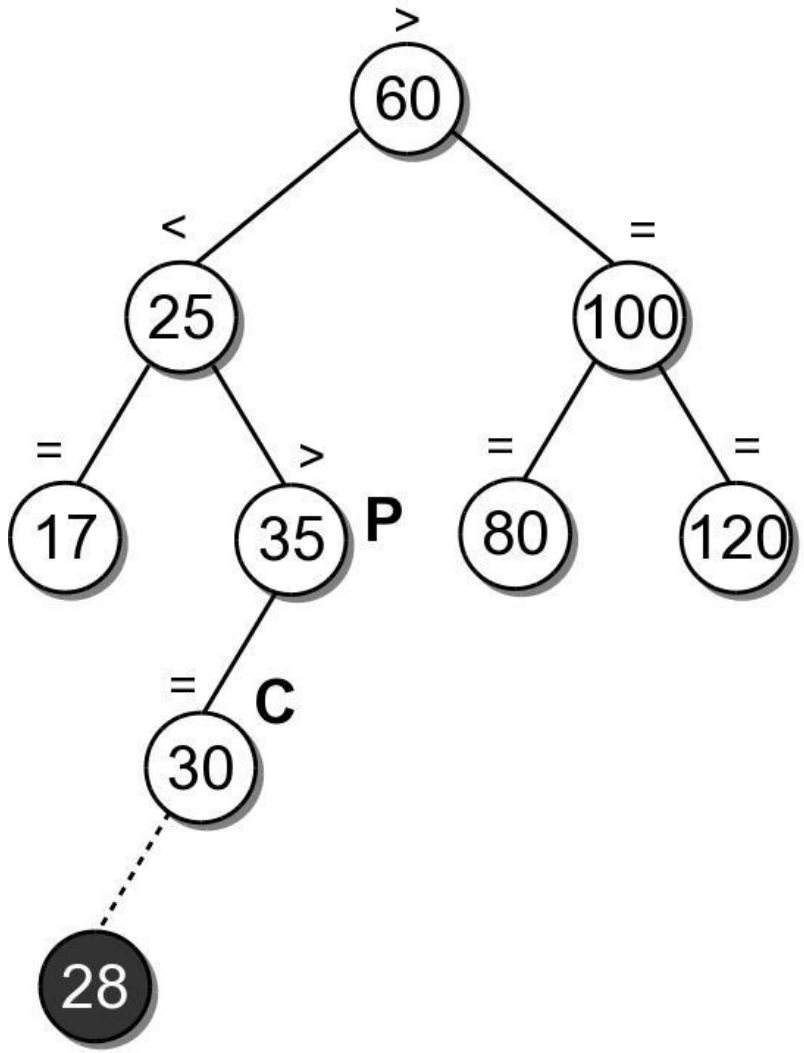
- Developed by G.M. **A**delson-**V**elskii and E.M. **L**andis in 1962
- Goal: try to keep the tree balanced during insertion and removal.
- Ensures height never exceeds 1.44 log n
- Worst case of tree height O(log n)
- An *AVL tree* is a binary search tree that the height of two children differ by no more than one.

These are AVL trees, ‘=‘ denotes children are of equal height, ‘<‘ or ‘>’ differ by one.



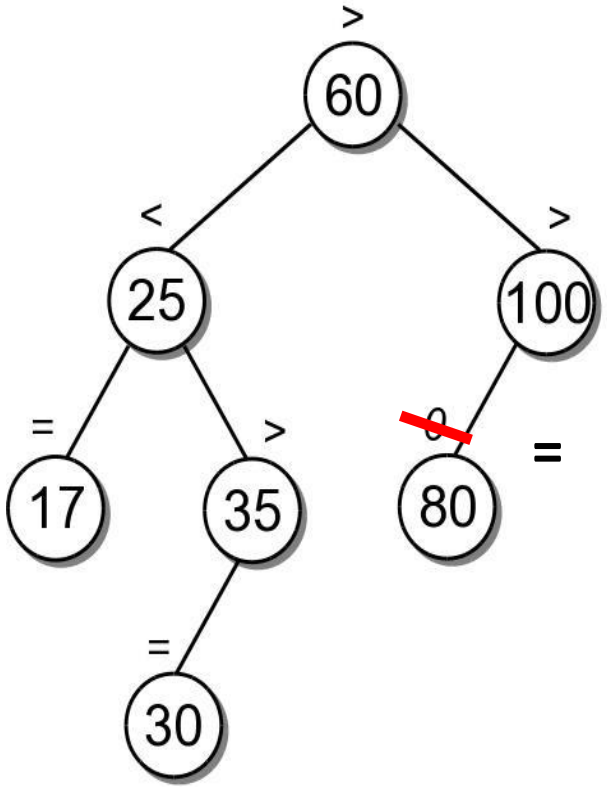


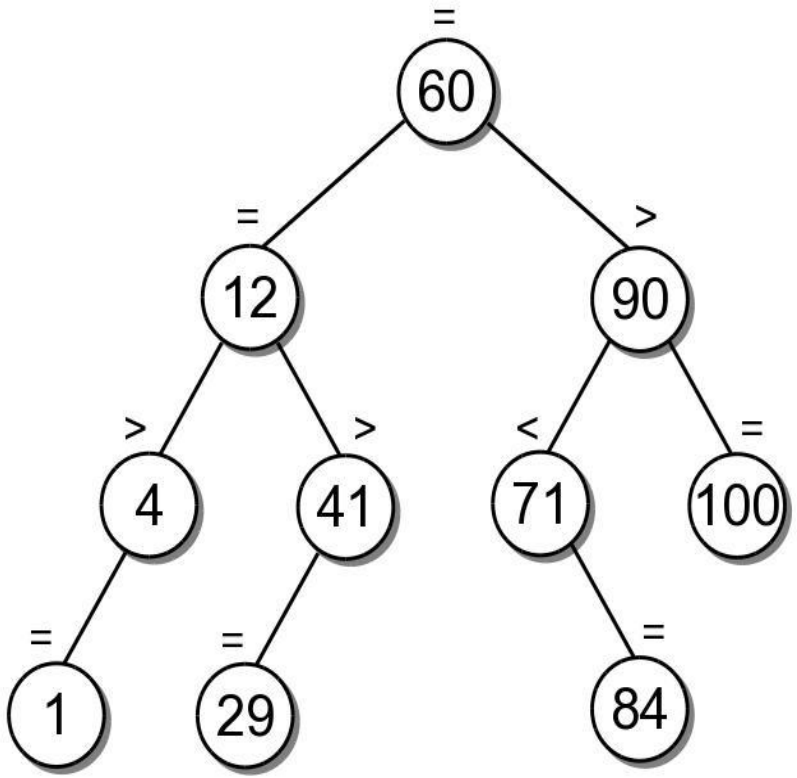
The tree on the right is **not** an AVL tree, after the node with 28 is inserted. The balance signs along the branch where 28 is inserted have to be revised. And the tree has to be re-balanced if to maintain it as an AVL tree.



Balance Factor

- So what states of balance exist?
  - Left-high ‘>’
  - Equal-high ‘=‘
  - Right-high ‘<‘



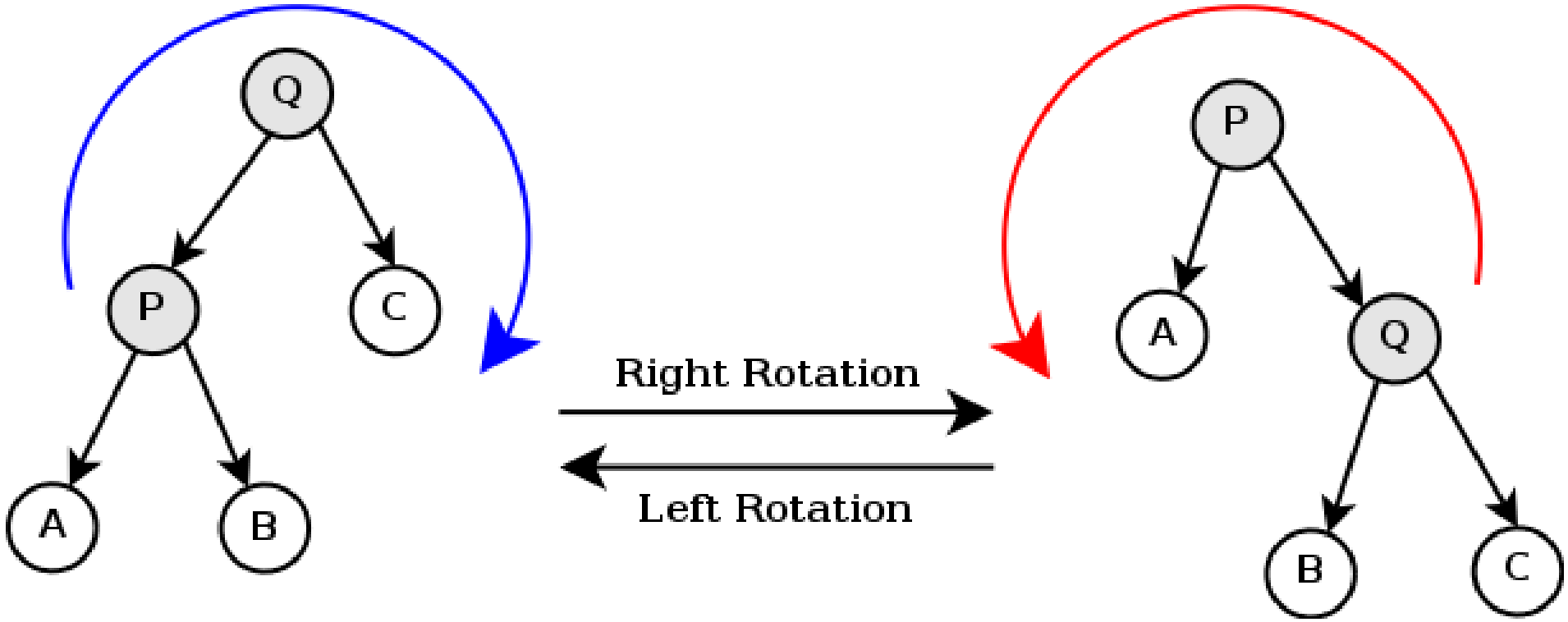


How do we maintain balance?

- Well that is the million dollar question!
- We will use *rotations*!
- All rotations happen at a fixed point
- This fixed point is the *pivot node*
  - Not really the instance of the node, but the location of the node!

Rotations

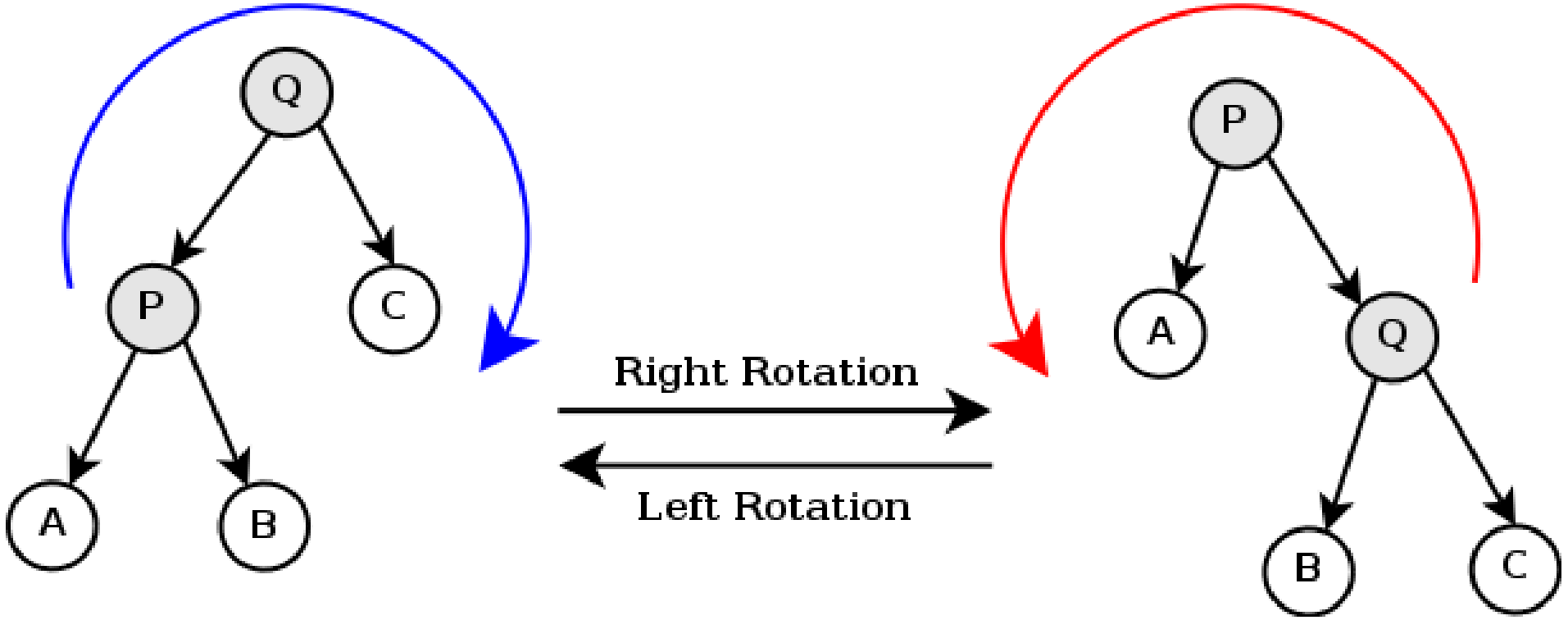
- Right Rotation
  - Pivot Q
  - Q becomes the parent of B
  - P becomes the parent of Q
- Left Rotation
  - Pivot P
  - P becomes the parent of B
  - Q becomes the parent of P



Rotations

```
def rotate_right( pivot ):
    # pivot == Q
    X = pivot.left          # Save P
    pivot.left = X.right    # Move B
    X.right = pivot         # Move Q
    return X
```

```
def rotate_left( pivot ):
    # pivot == P
    X = pivot.right         # Save Q
    pivot.right = X.left    # Move B
    X.left = pivot          # Move P
    return X
```

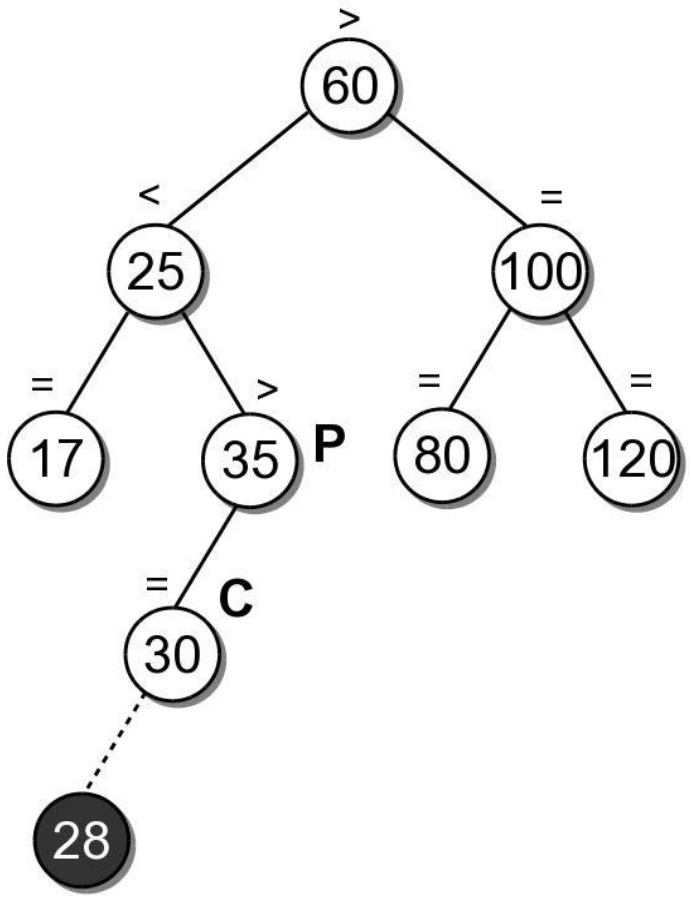


Insertion

- Now we have our weapon (rotations) and we will use it
- Step 1. Find a place to add the element as a leaf node
  - Same as BST
- Step 2. Rebalance
  - Four Cases to consider

Case 1.

- Add to the left side of a subtree that is already left-high

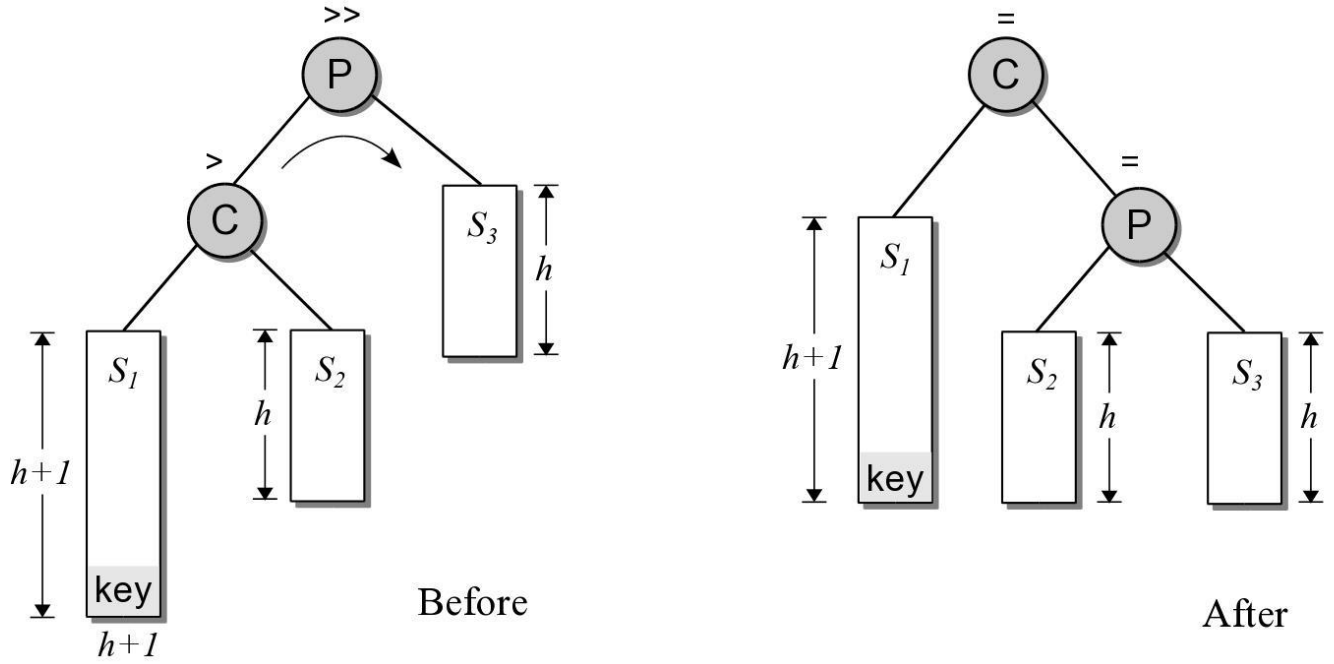


Insert 28.

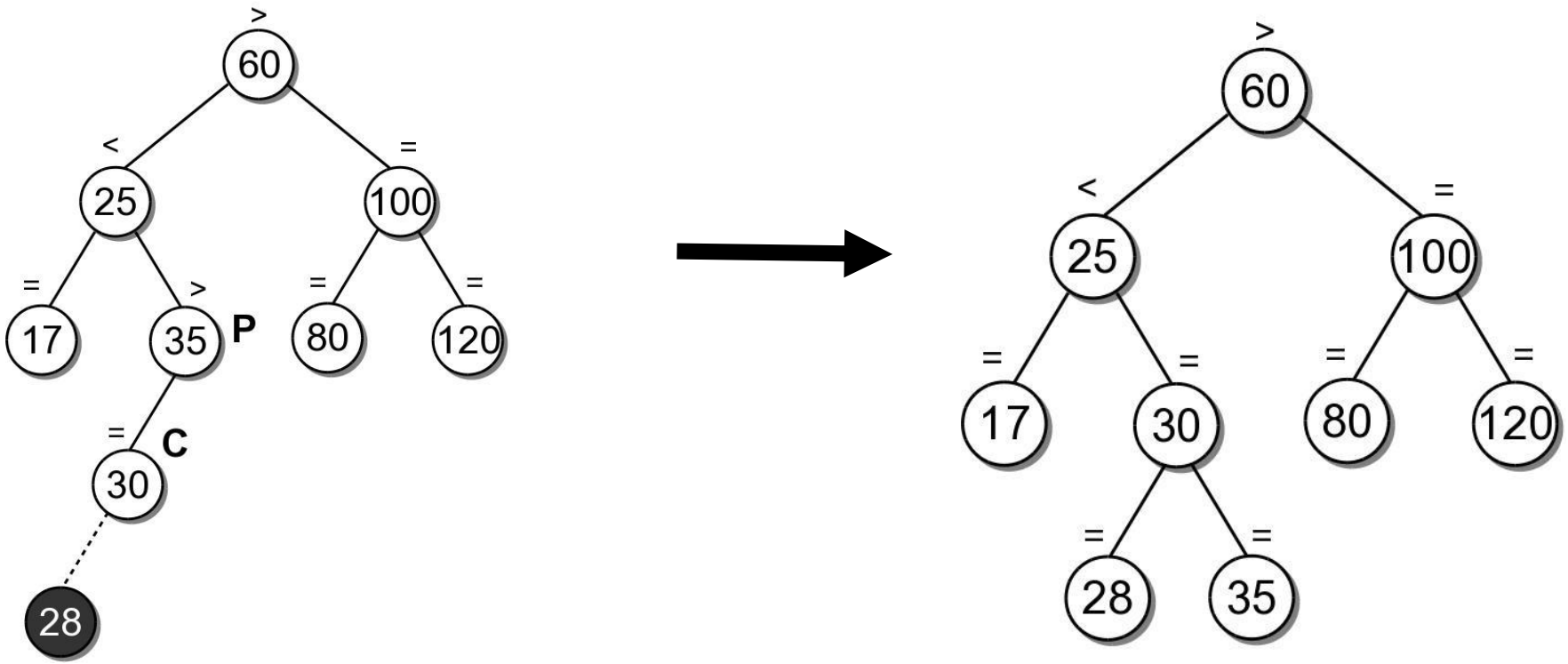
1. **Walk down to find a place to insert 28.**
2. **Walk back up the path and found a node with left-high and we added to the left! Now very left-high! We need to fix this!**
3. **Fix?**

Case 1

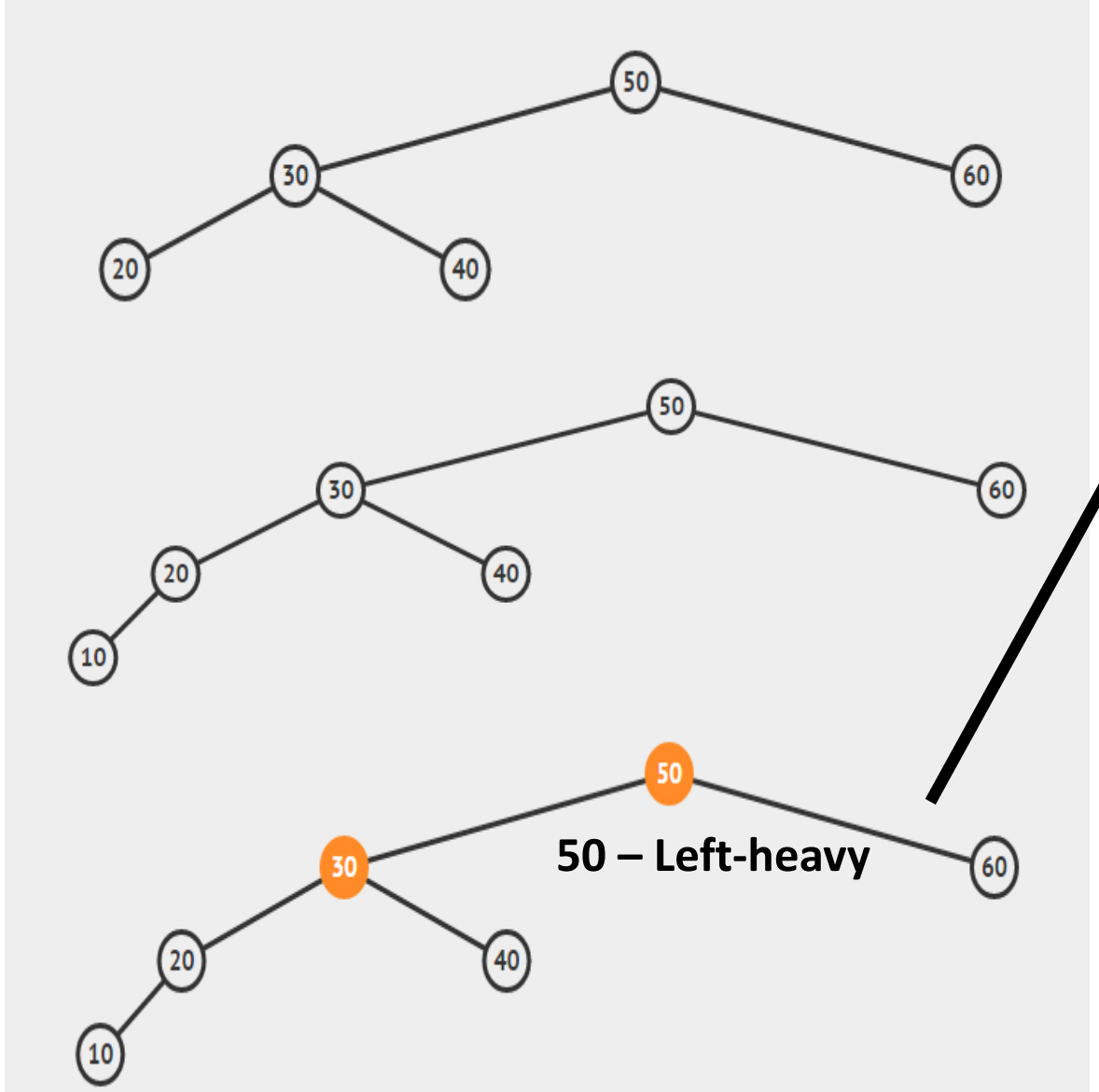
- How do we fix the balance?
- At the node with left-high (P) on the return path,….
- What do we know about the subtree P?
  - We know the left side subtree height is now 2 nodes higher than the height of the right subtree
  - We know if we could rotate one node for the left into the right the subtrees will be equal
- Therefore, rotate right!

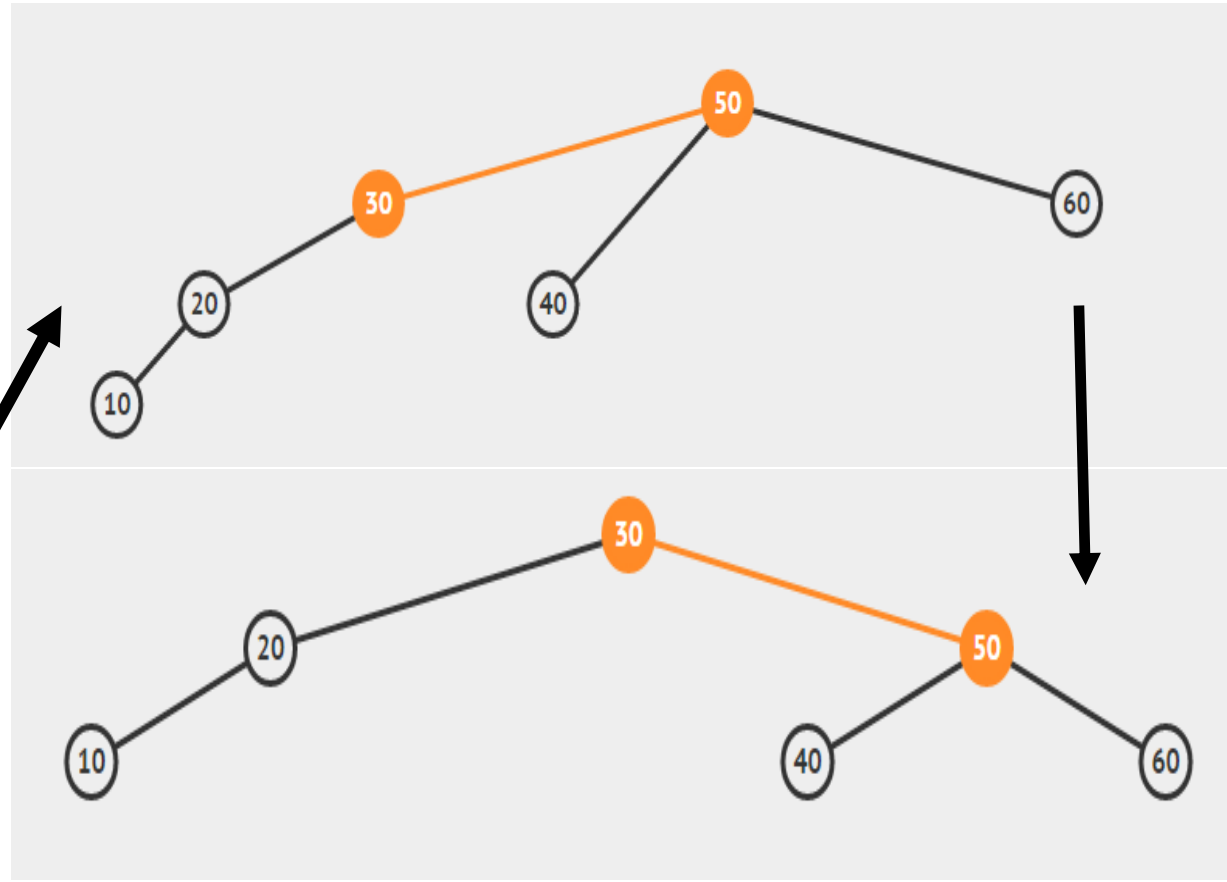


Easy Example (Case 1)



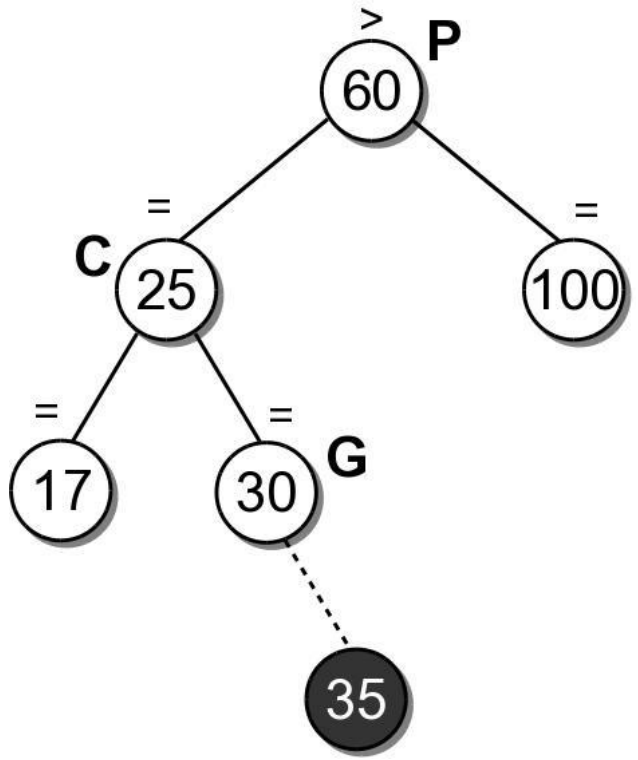
More Complex Example (Case 1)





Case 2

- Add node as a right or left leaf node of a path where you are right of your grandparent, C, (equal-high) and left of your great grandparent, P, (left-high).

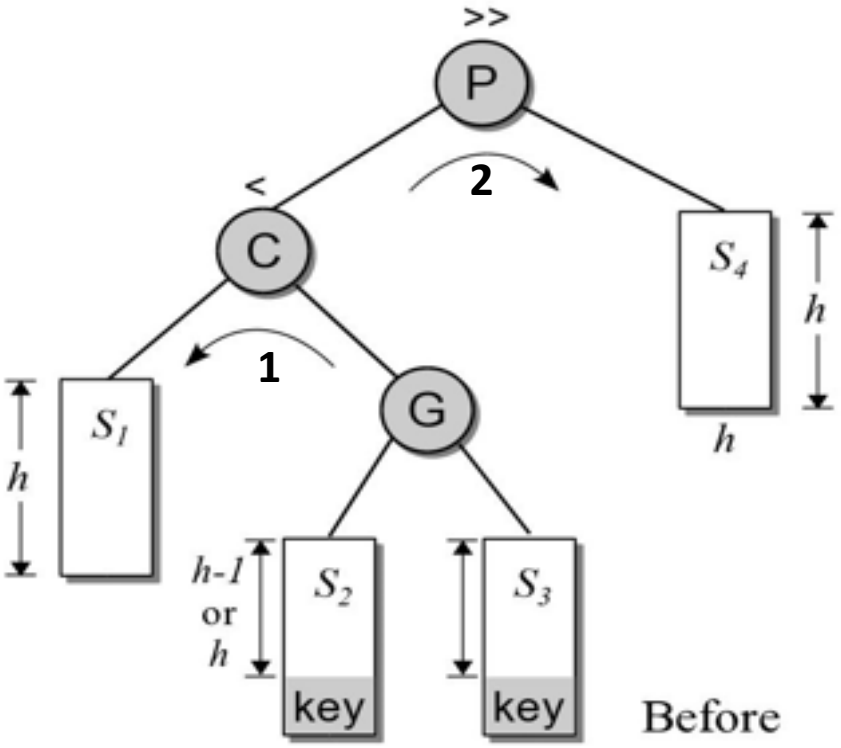


Insert 35.

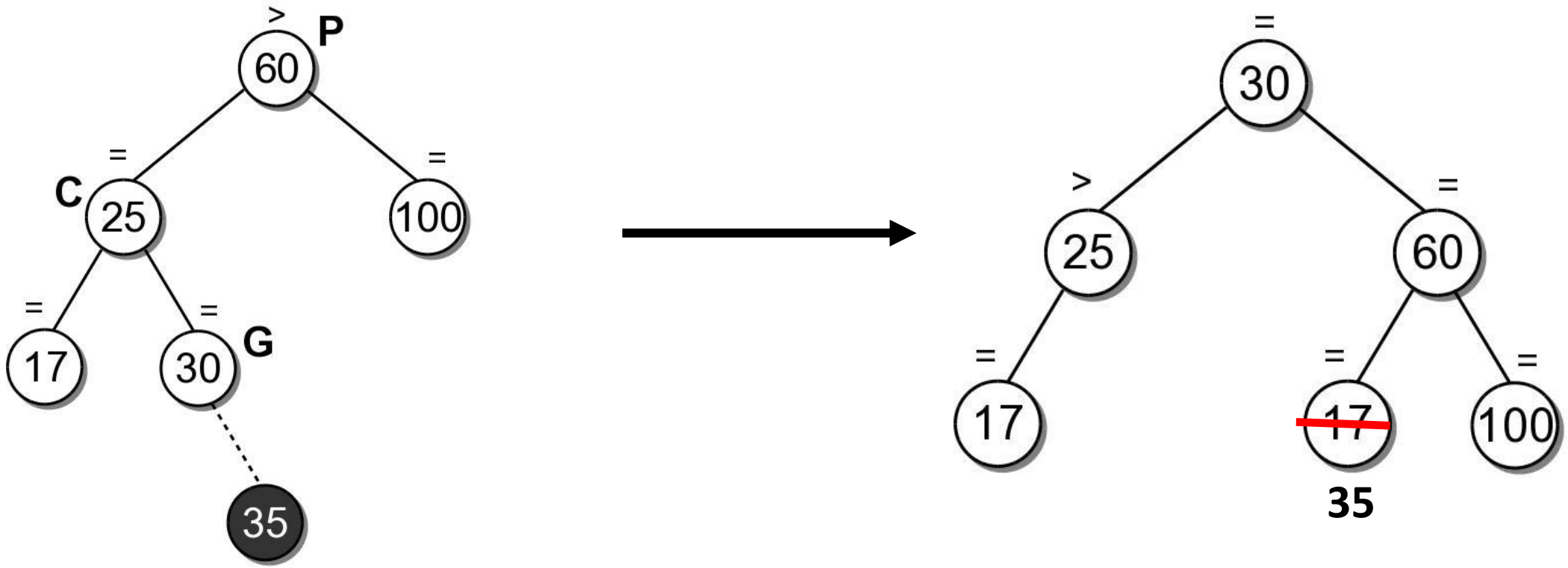
1. **Walk down to find a place to insert 35.**
2. **Walk back up the path to 30, make G right-high ‘<‘**
3. **Walk back up the path to 25, make C right-high ‘<‘**
4. **Walk back up the path to 60, Oh no! Was already left-high and now left is even more taller! Fix this!**
5. **Fix?**

Case 2

- How do we fix this imbalance?
- At node with left-high (P)
- Would a simple rotation help?
  - Left rotation around P – would make the left branch of P more taller (more ‘left-high’)!
  - A simple right rotation around P – would make C the new root, P the right child of C, G the left child of P, s1 the left child of C. The tree is not balanced as the right-height(C)= height(P)+1 = height(G)+h+1 >> left-height(C) == h
- What if we do a left rotation on the left child of P (node C) first, followed by a right rotation of P?
- Result
  - 1. Left rotation on P.left (C in this case)
  - 2. Right rotation on P

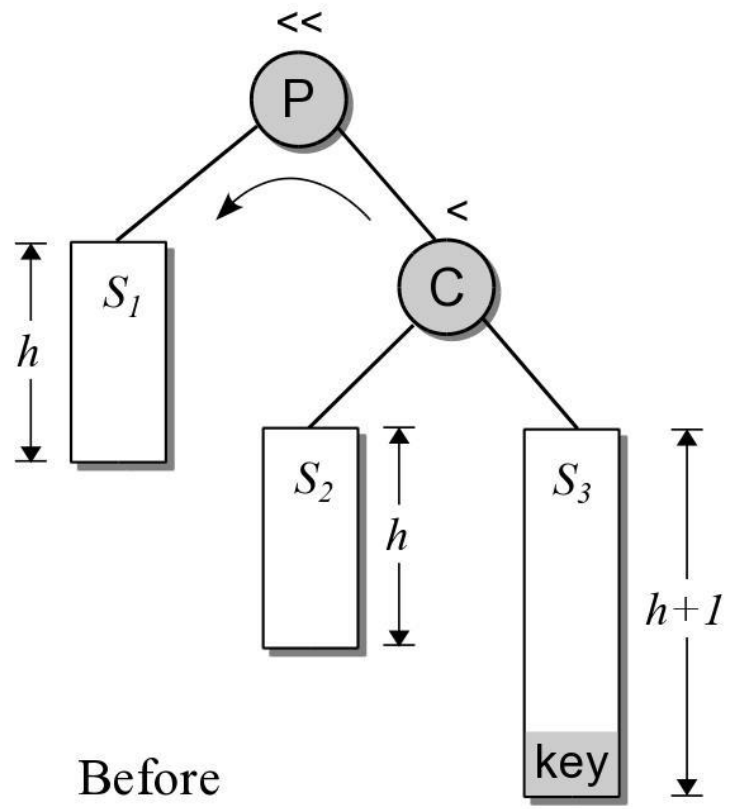


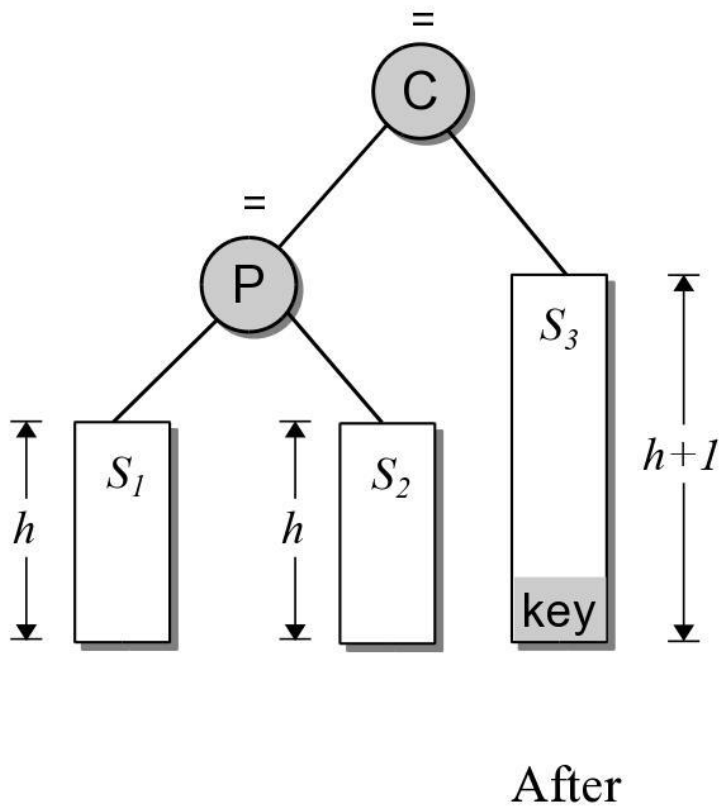
Example (Case 2)



Case 3

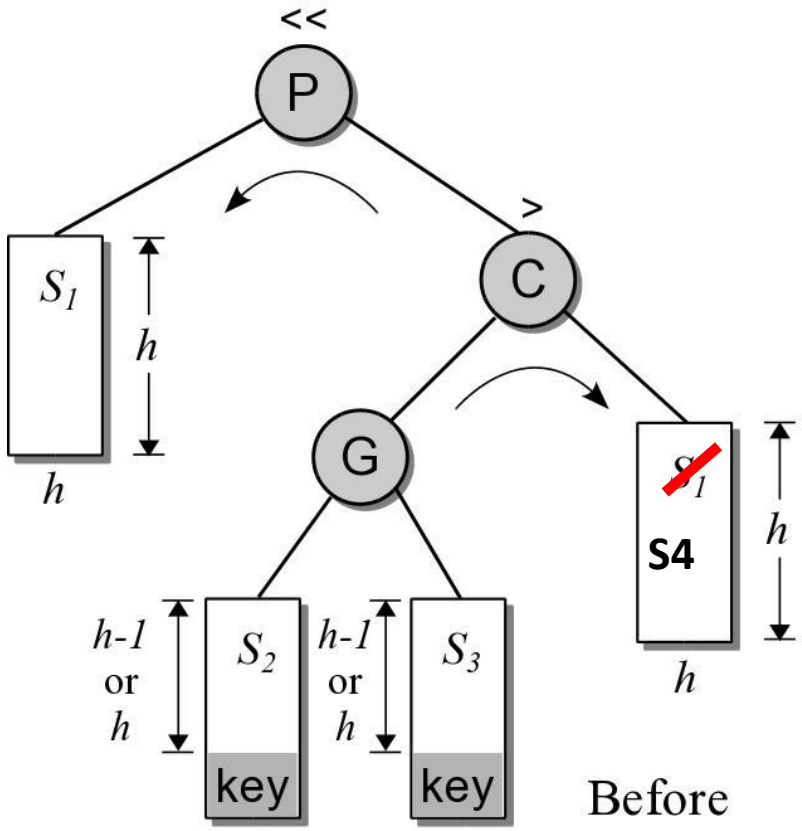
- Mirror image of Case 1
- P is right-high
- New key is inserted in right subtree of C
- Shown below is ‘before’ and ‘after’ balancing, but after inseration!

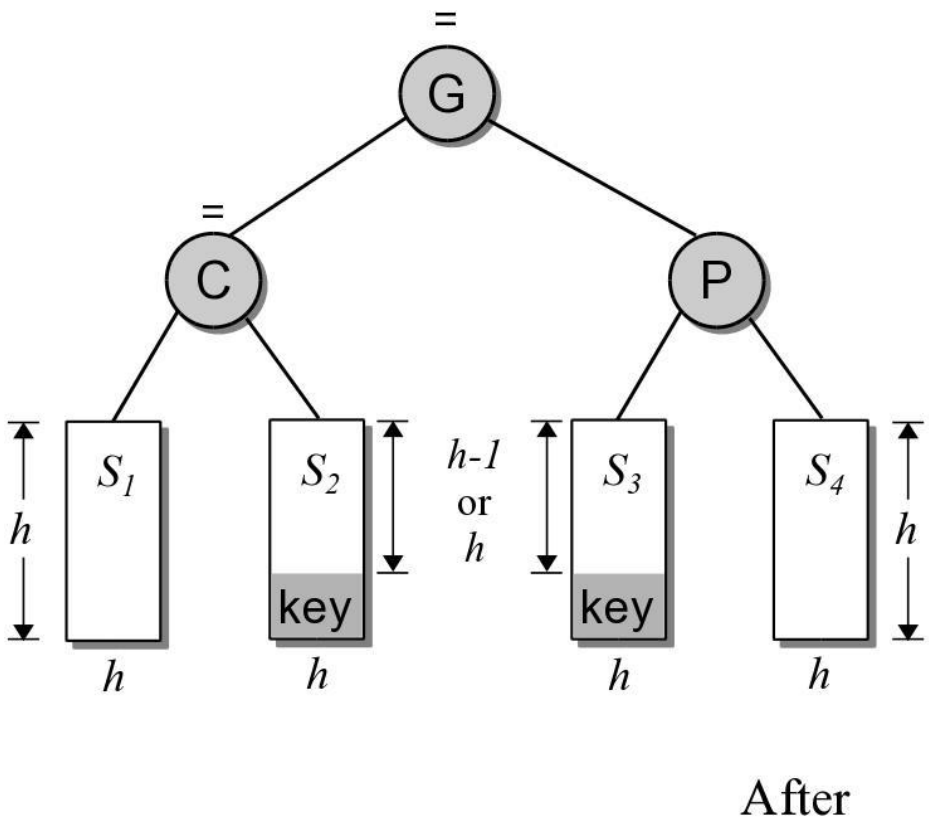




Case 4

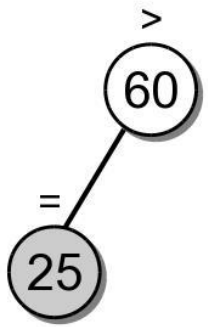
- Mirror image of Case 2
- P is right-high
- G is the left child of C instead of the right
- Right child of C is S4



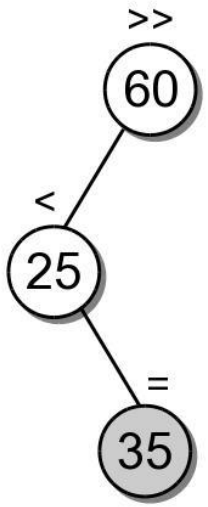


Building AVL Tree [60,25,35,100,17,80]

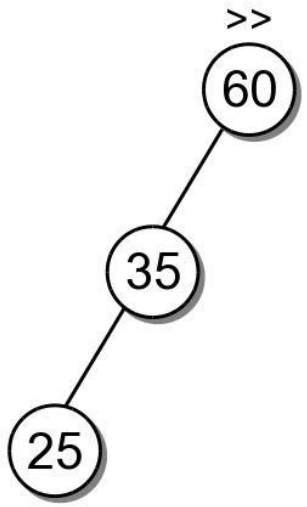
(a) Insert 60.



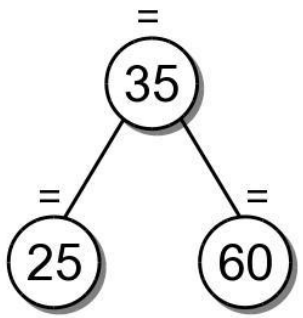
(b) Insert 25.



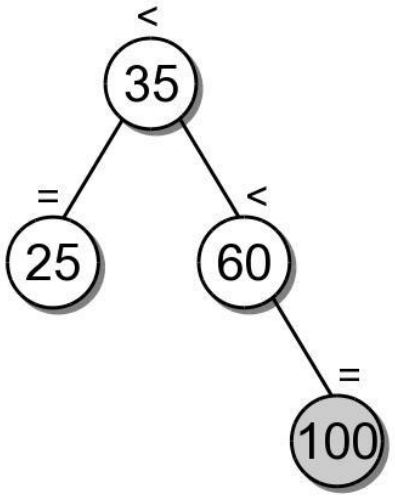
(c) Insert 35.



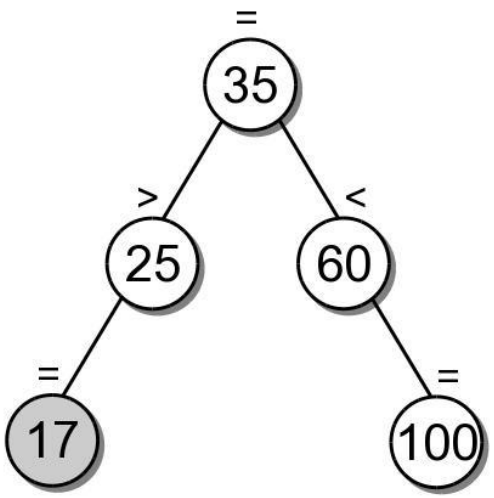
(d) Left rotate at 25.



(e) Right rotate at 60.

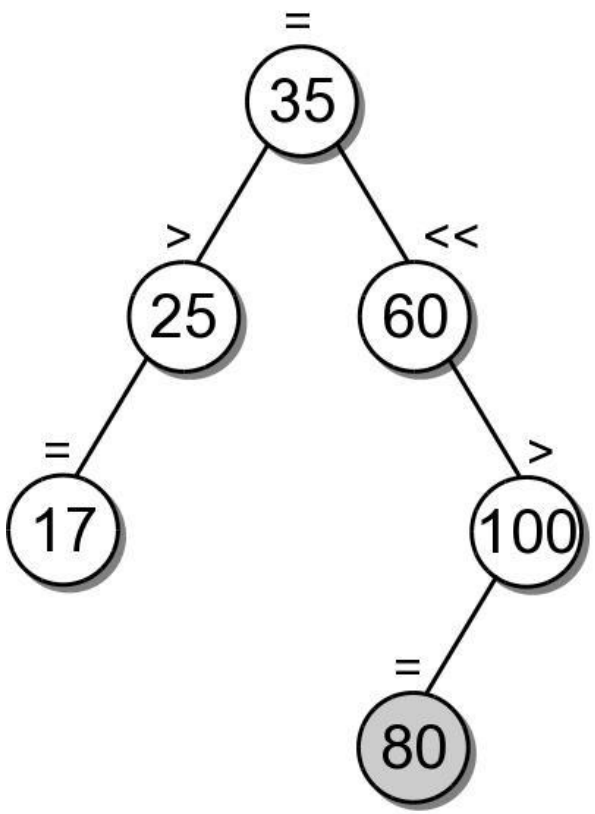


(f) Insert 100.

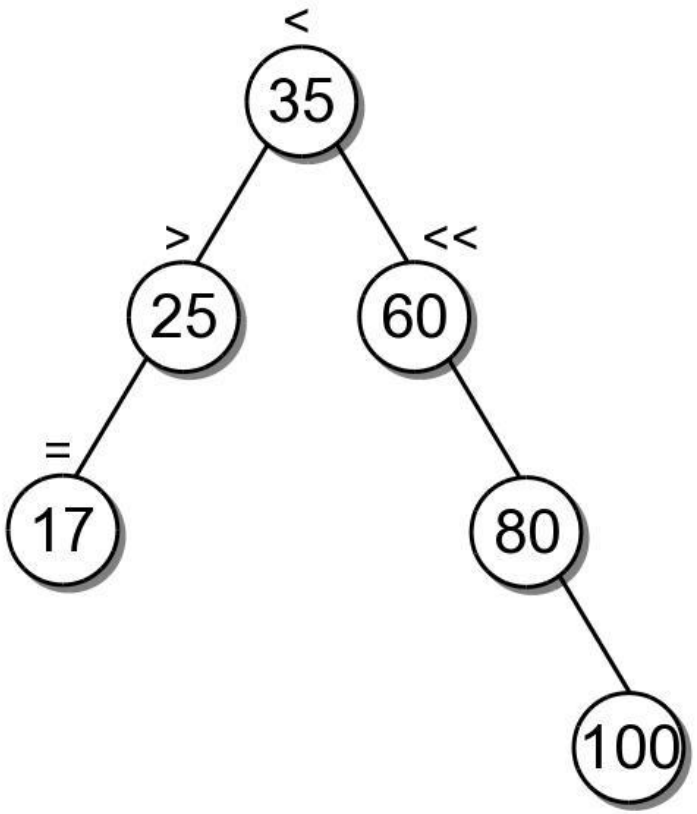


(g) Insert 17.

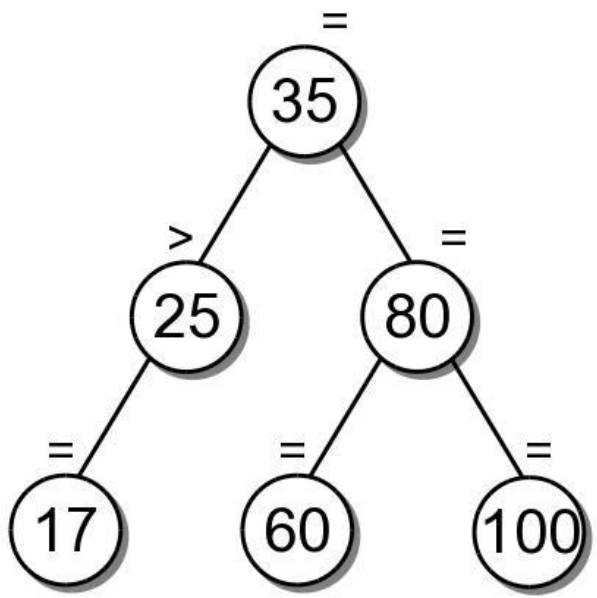
Building AVL Tree [60,25,35,100,17,80]



(h) Insert 80.



(i) Right rotate at 100.



(j) Left rotate at 60.

AVL Removal

- First remove as we do in BST
- Rebalance