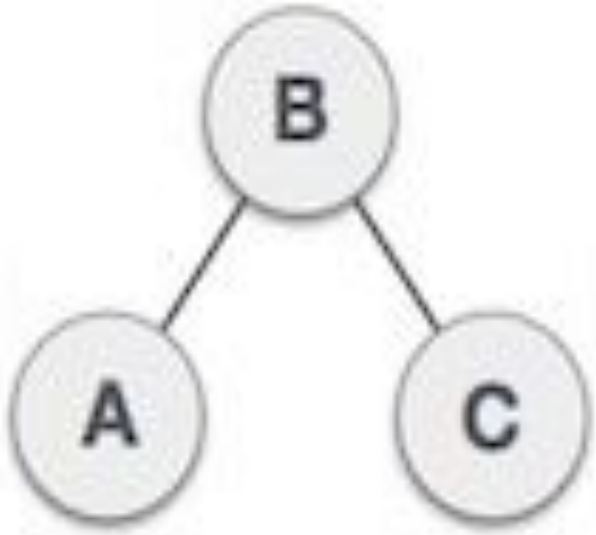
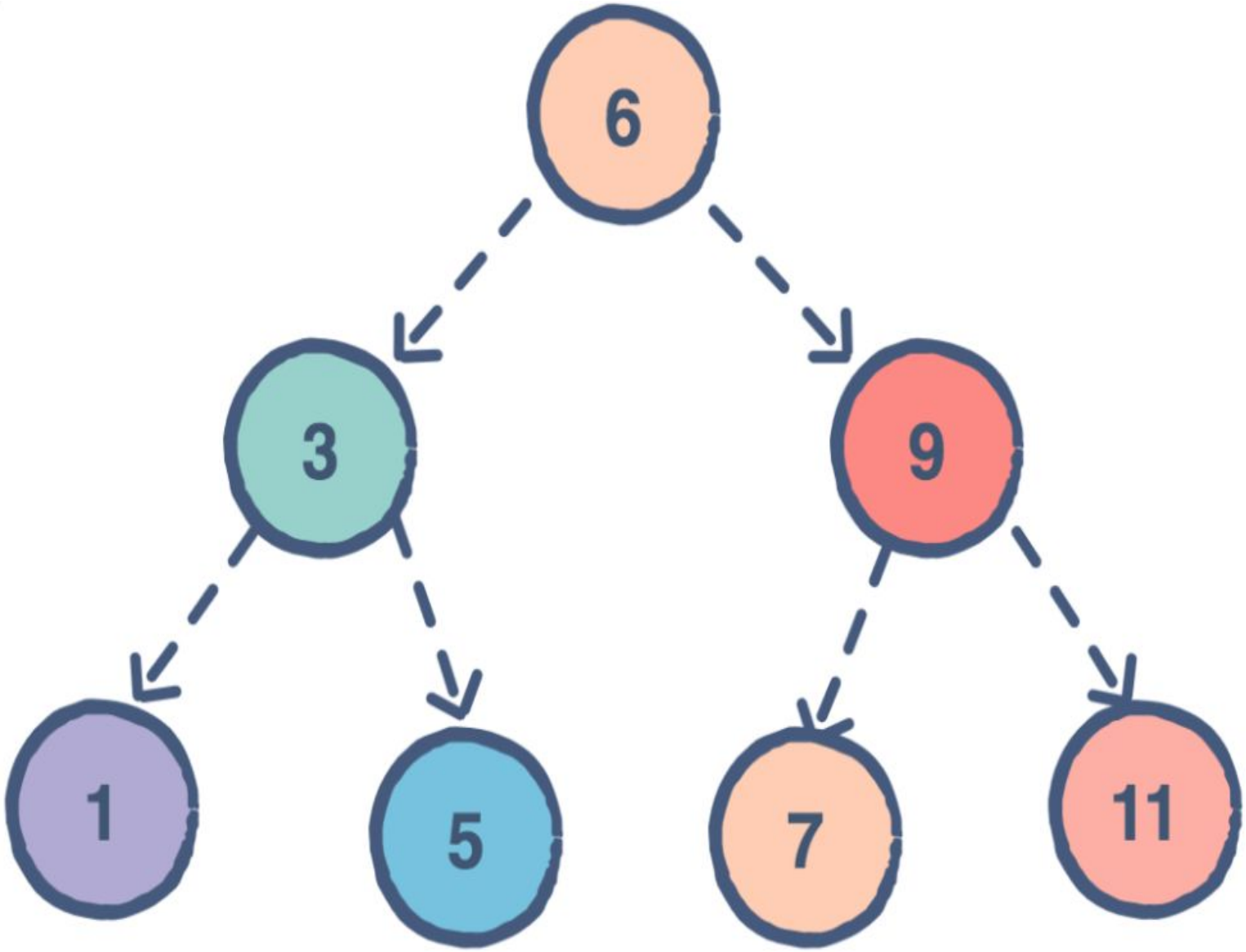
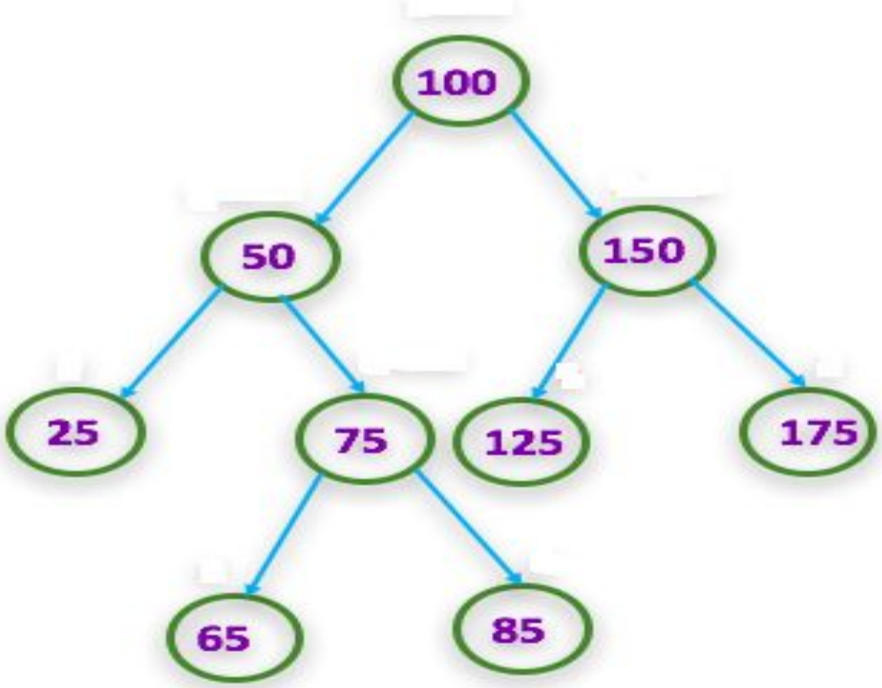
Balanced binary tree (or AVL Tree)

Is a binary tree in which the height of two subtrees of every node never differ by more than one.





An example of a balanced tree



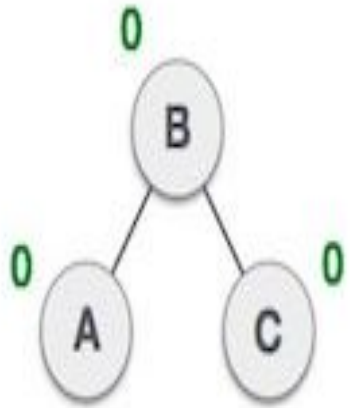
AVL Tree

The balance of a node in a an AVL tree is defined as:

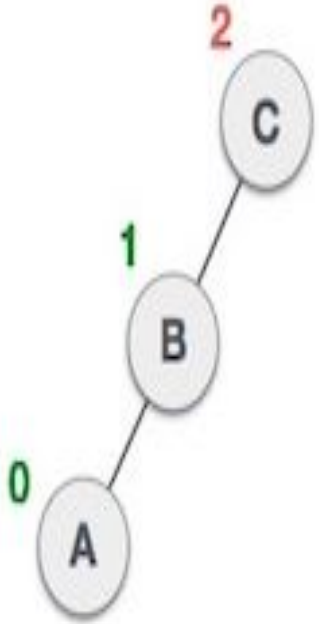
Balancing factor BF =Height of left subtree – height of right subtree

The value of balance factor should always be -1, 0 or +1.

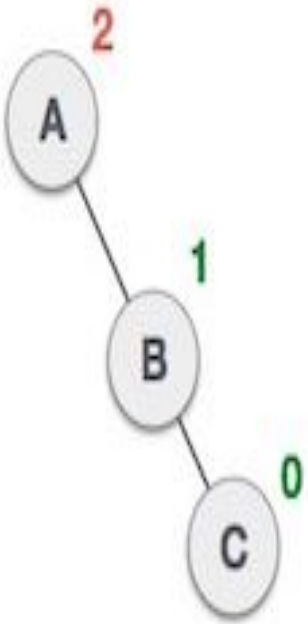
Two subtree are of same height, then BF =0
Left subtree is higher, BF =+1
Right subtree is higher, BF =-1



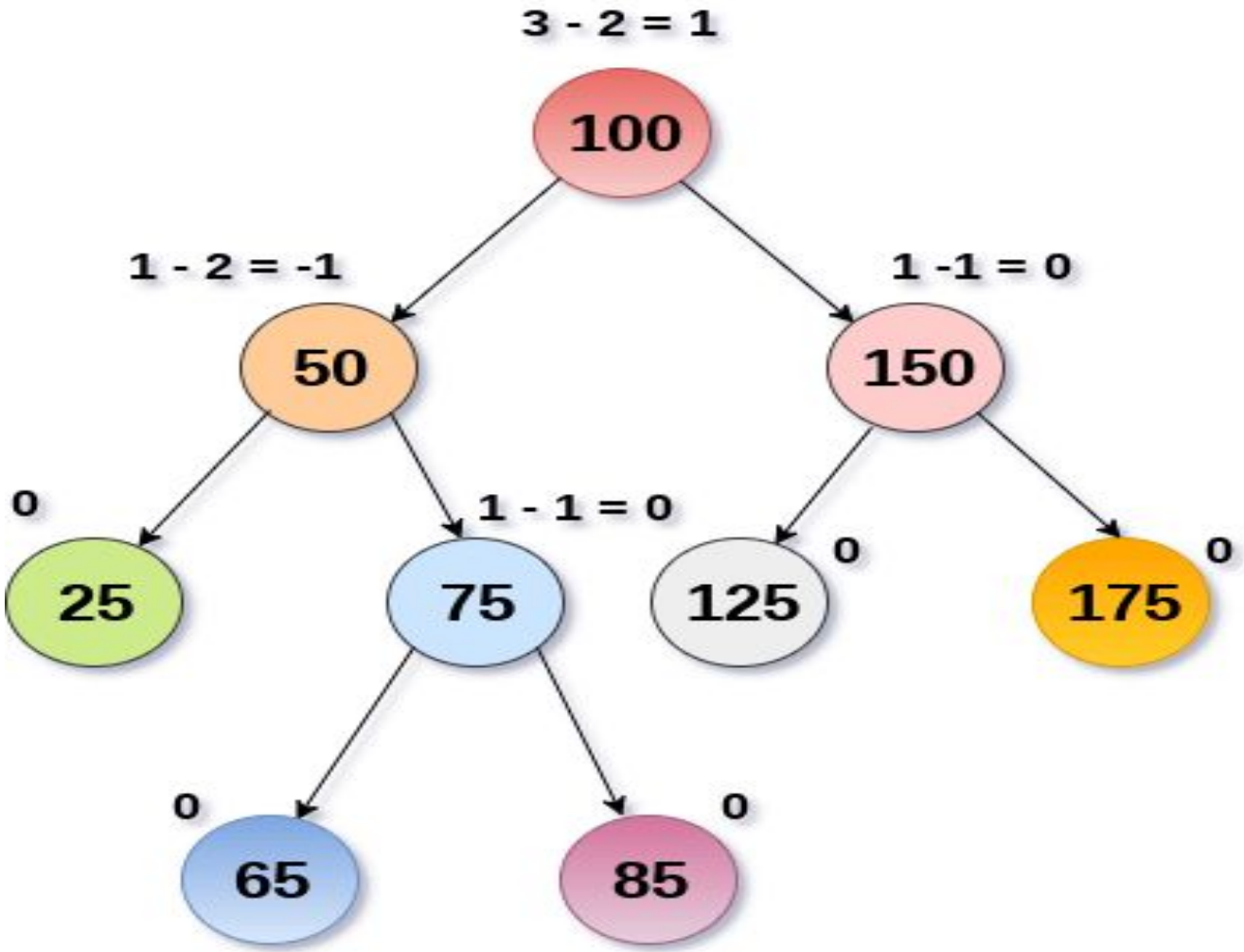
Balanced



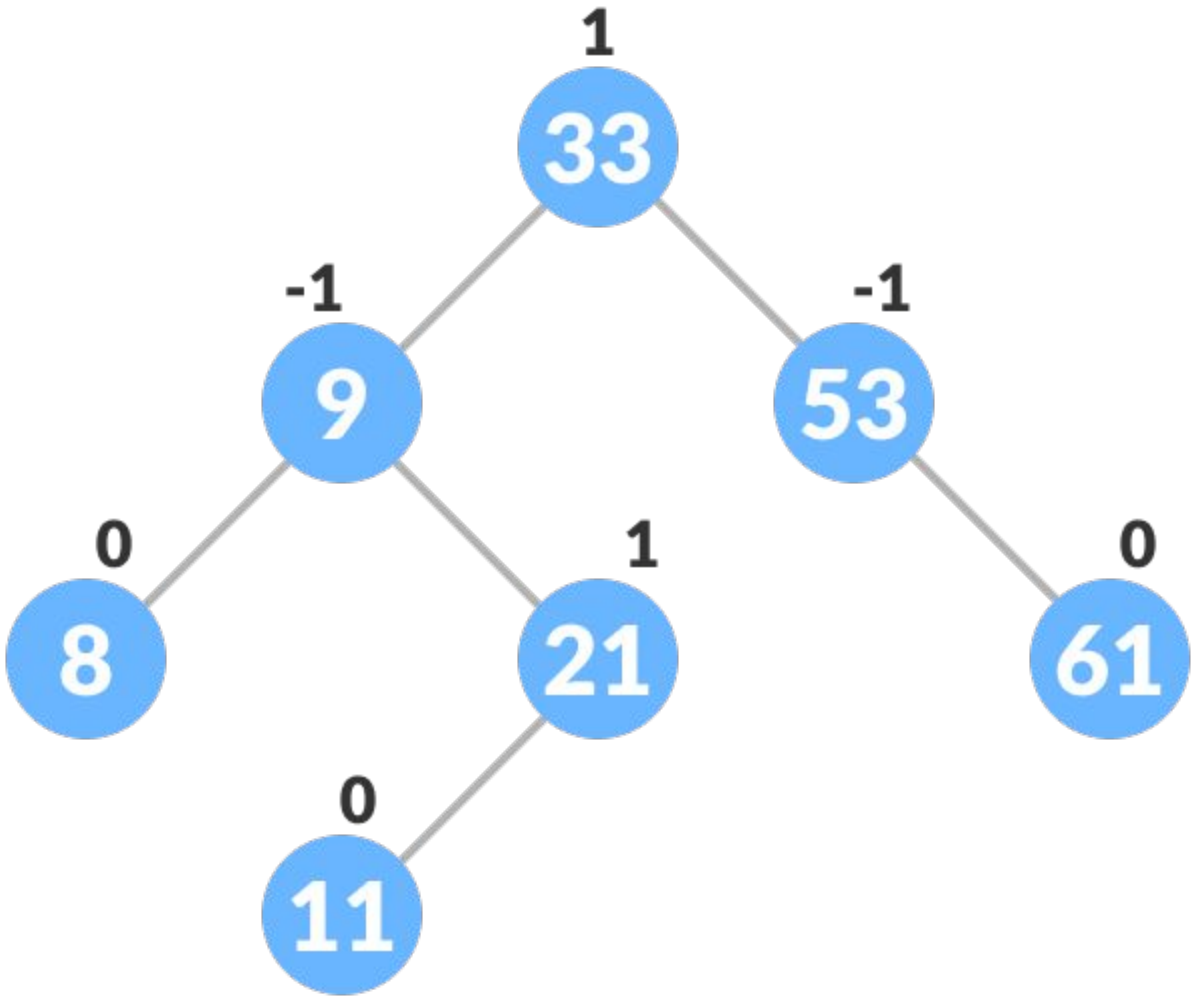
Not balanced

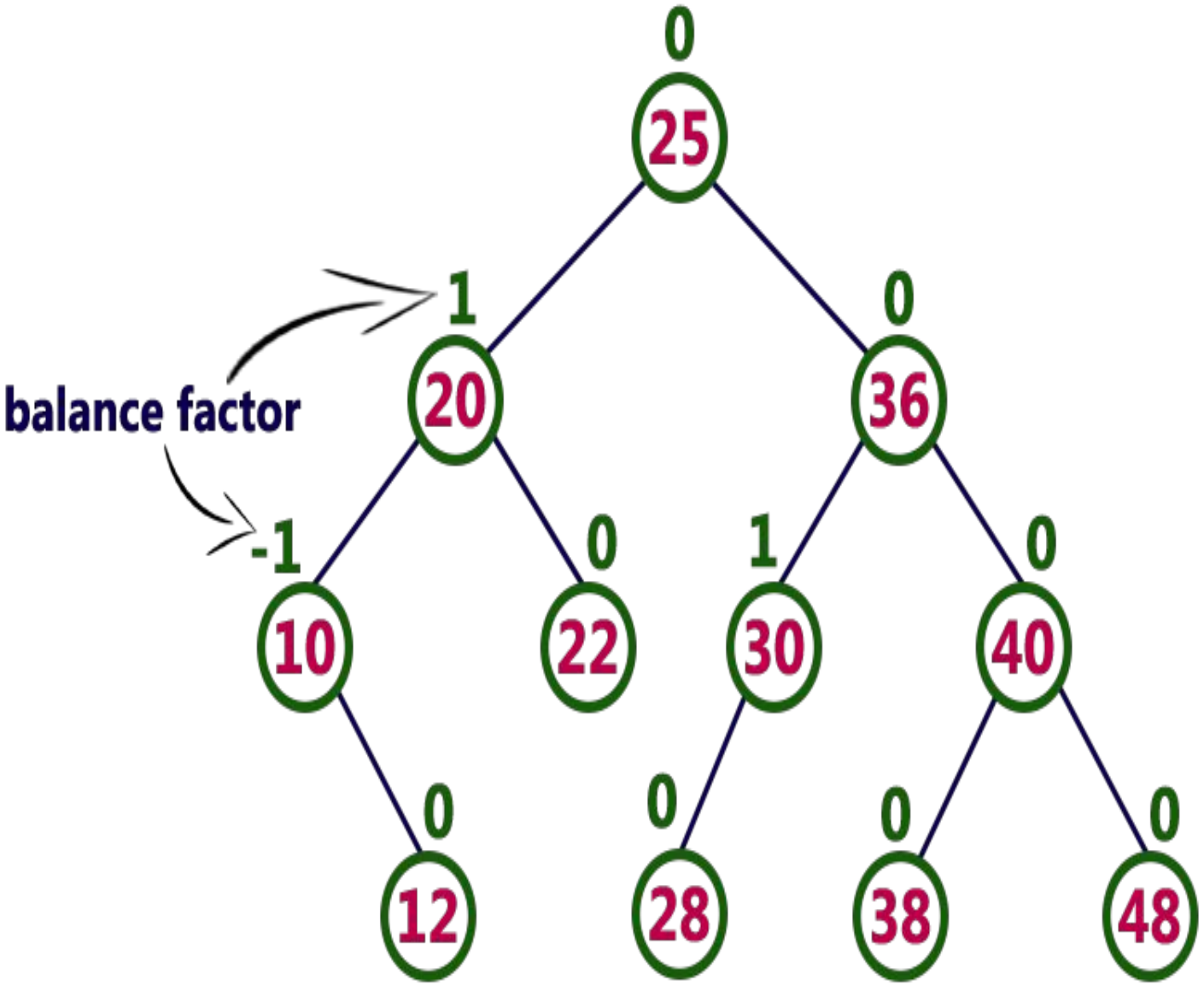


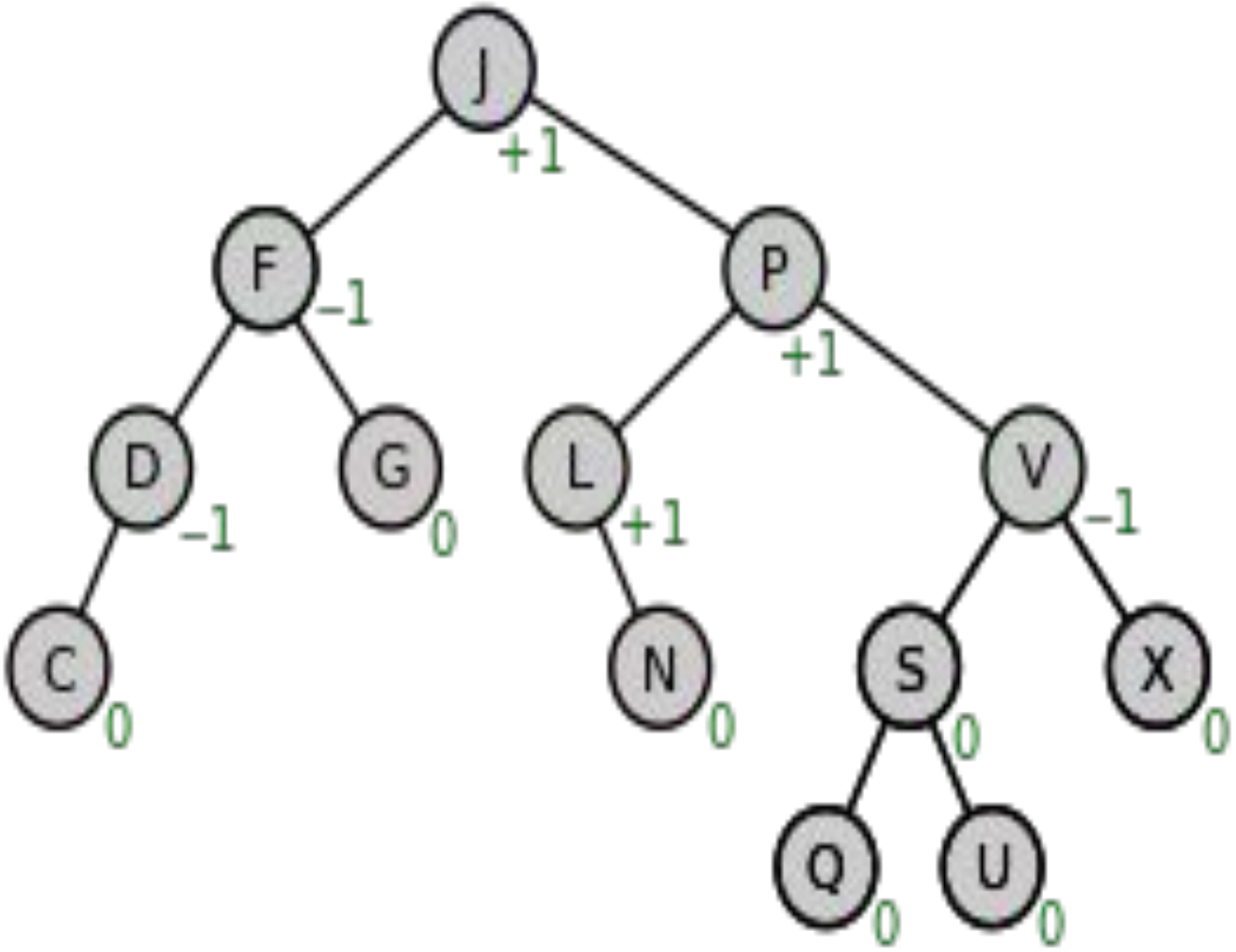
Not balanced



AVL Tree


1
33
-1
9
-1
53
0
8
1
21
0
61
0
11


balance factor
0
25
1
20
0
36
-1
10
0
22
1
30
0
40
0
12
0
28
0
38
0
48


J
+1
F
−1
P
+1
D
−1
G
0
L
+1
V
−1
C
0
N
0
S
0
X
0
Q
0
U
0

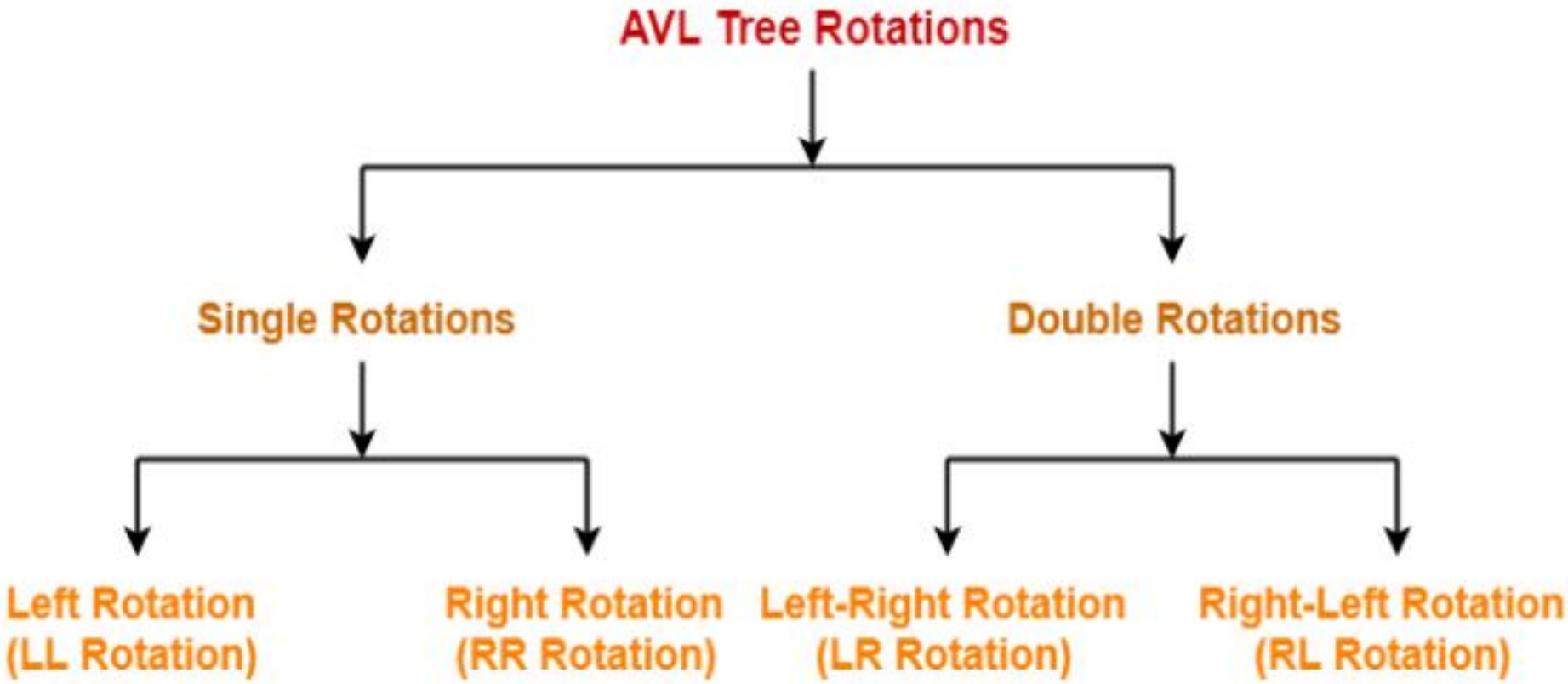
Create an AVL tree or Insert a node into an AVL tree

1. Insert the node as a node in the binary search tree

2. Compute the Balancing Factor
   If any node has a balancing factor -2 or 2, then a rotation is necessary.

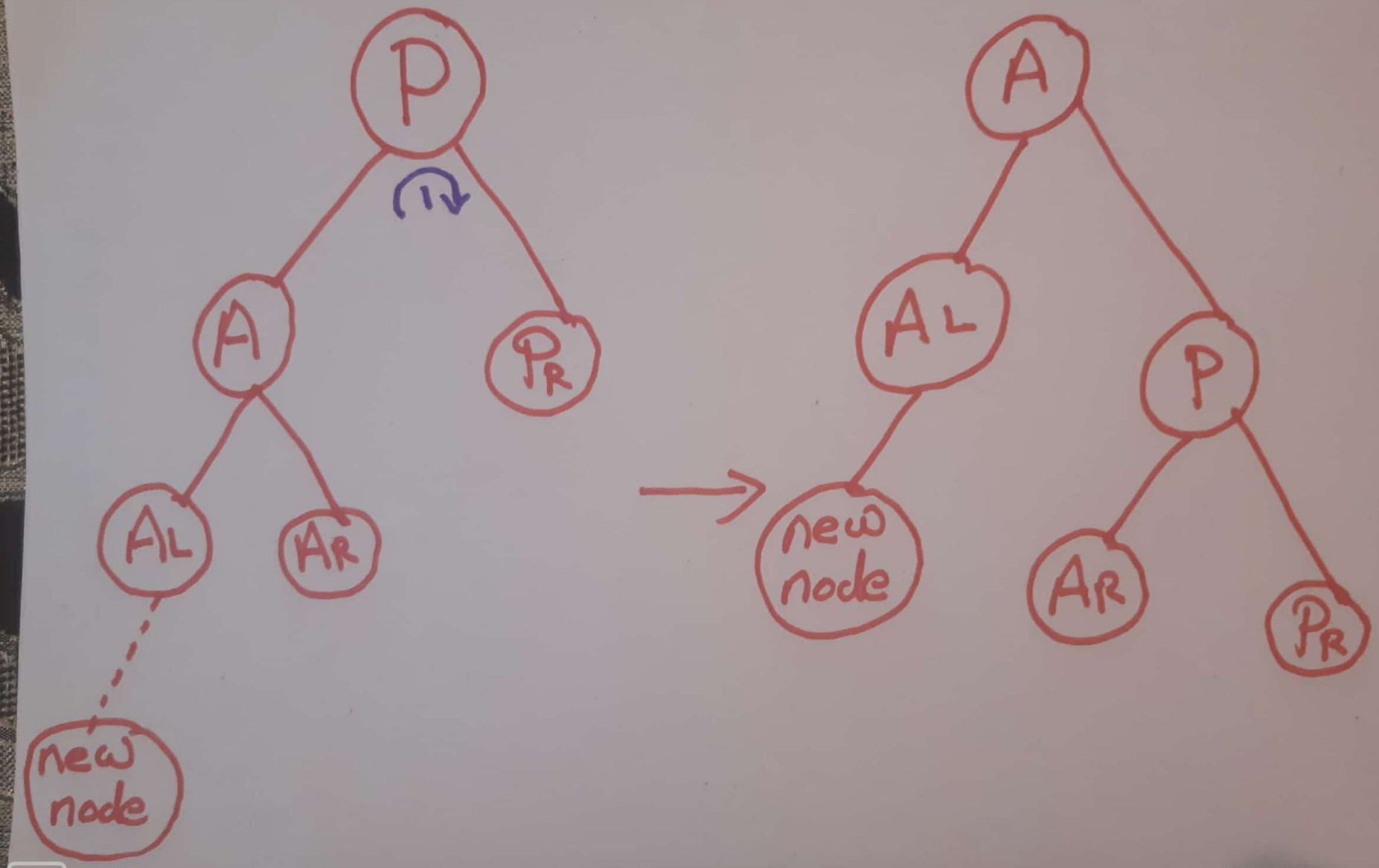
3. Decide the pivot node.
   Any node with absolute value 2 is the pivot node. If there are more than one node has a BF=2, then the nearest node to newly inserted node will be the pivot node.

4. Balance the unbalance tree.

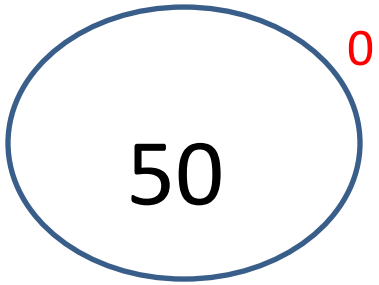


1. Left Rotation (LL Rotation) Unbalance due to left to left insertion
2. Right Rotation (RR Rotation) Unbalance due to left to left insertion
3. Left-Right Rotation (LR Rotation)
4. Right-Left Rotation (RL Rotation)

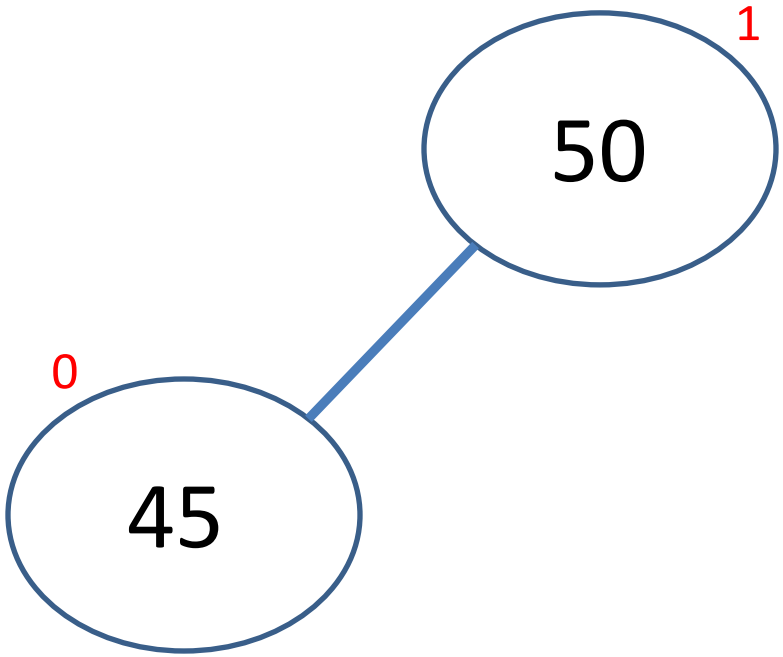
L.L



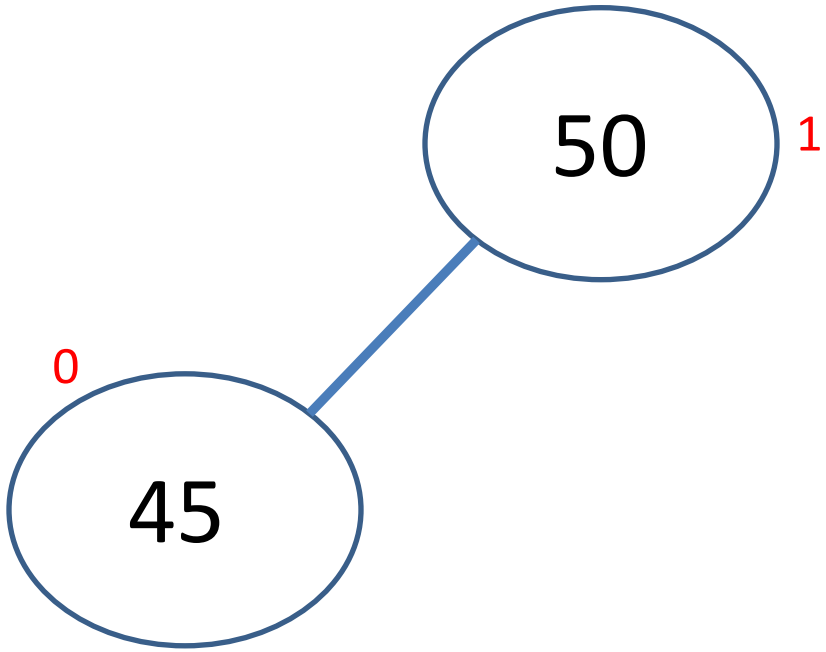
Insert 50



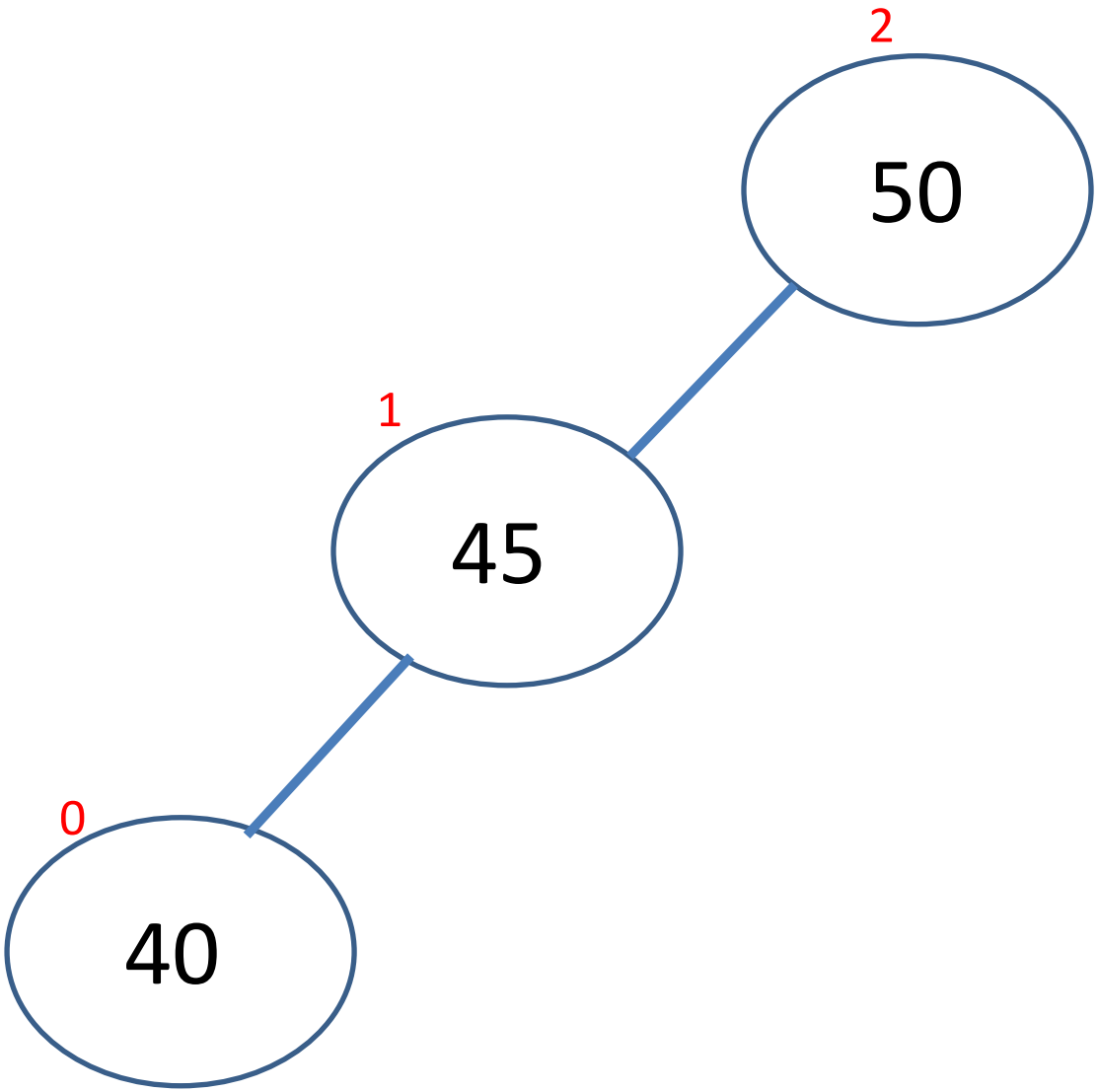
Insert 45



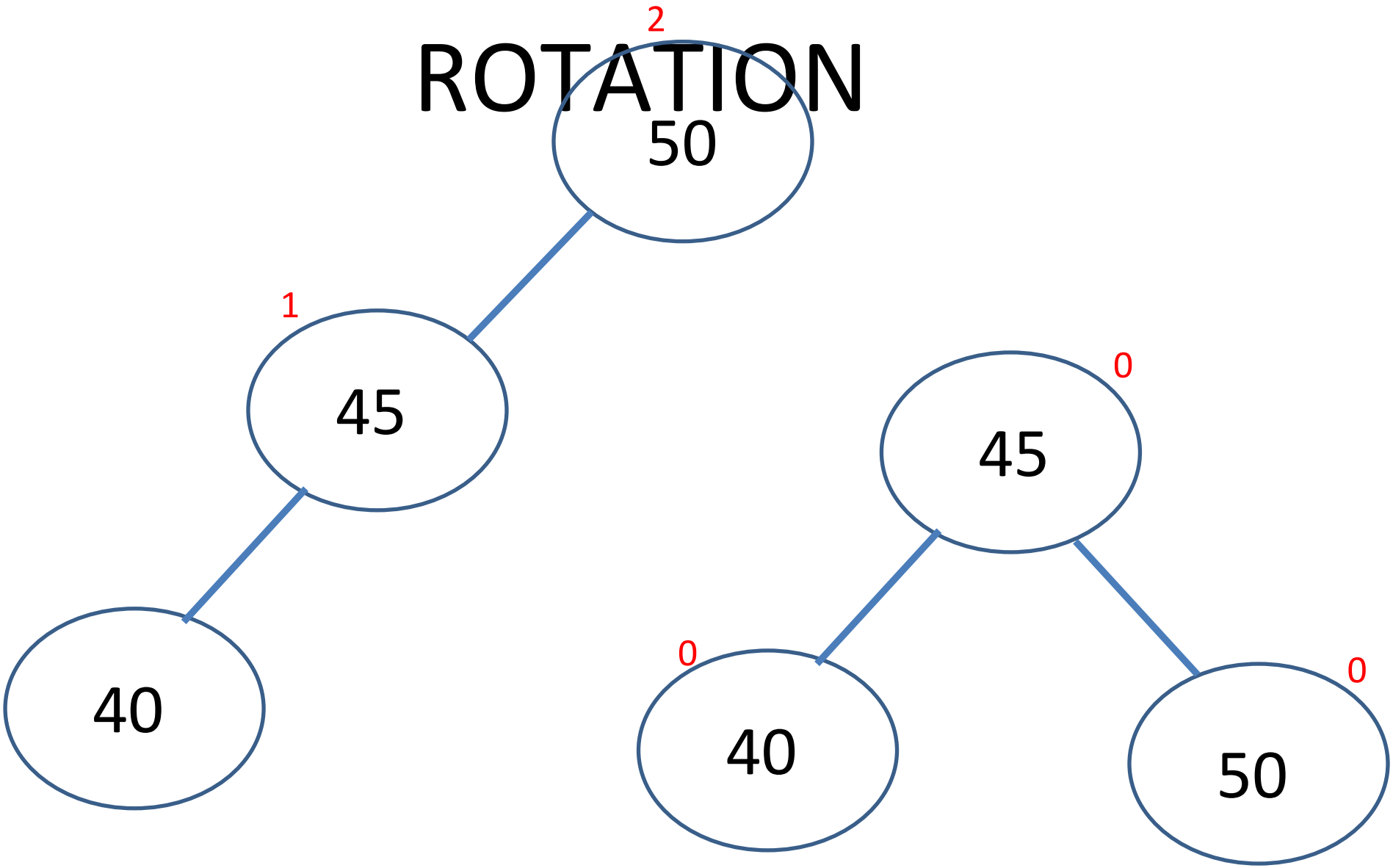
Insert 45



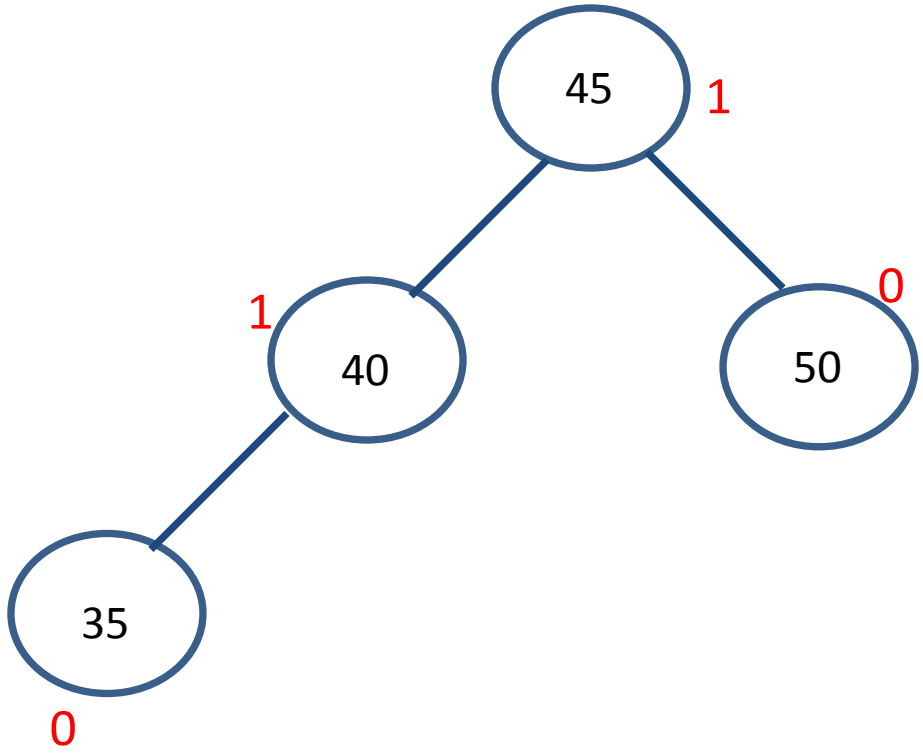
Insert 40

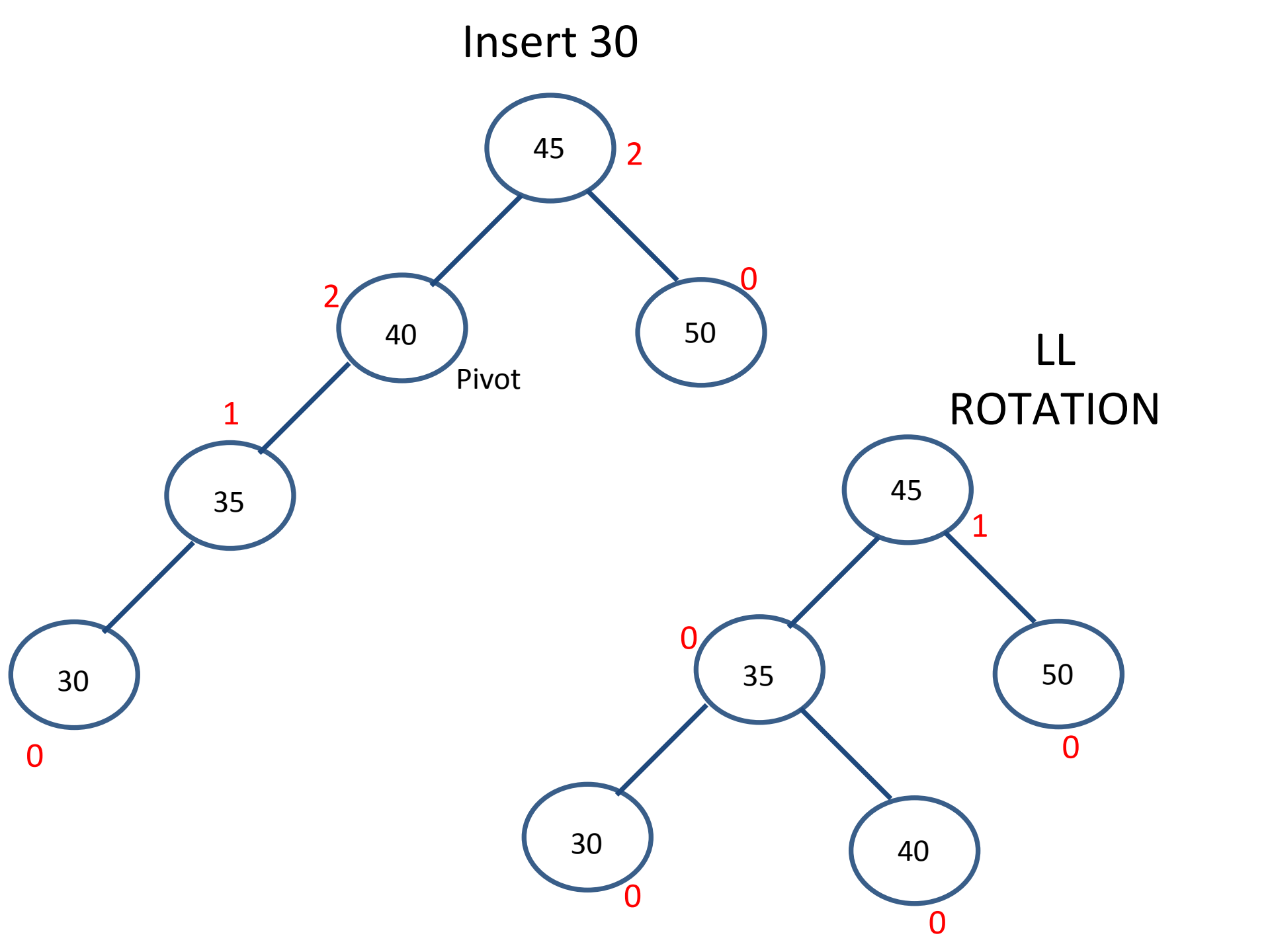


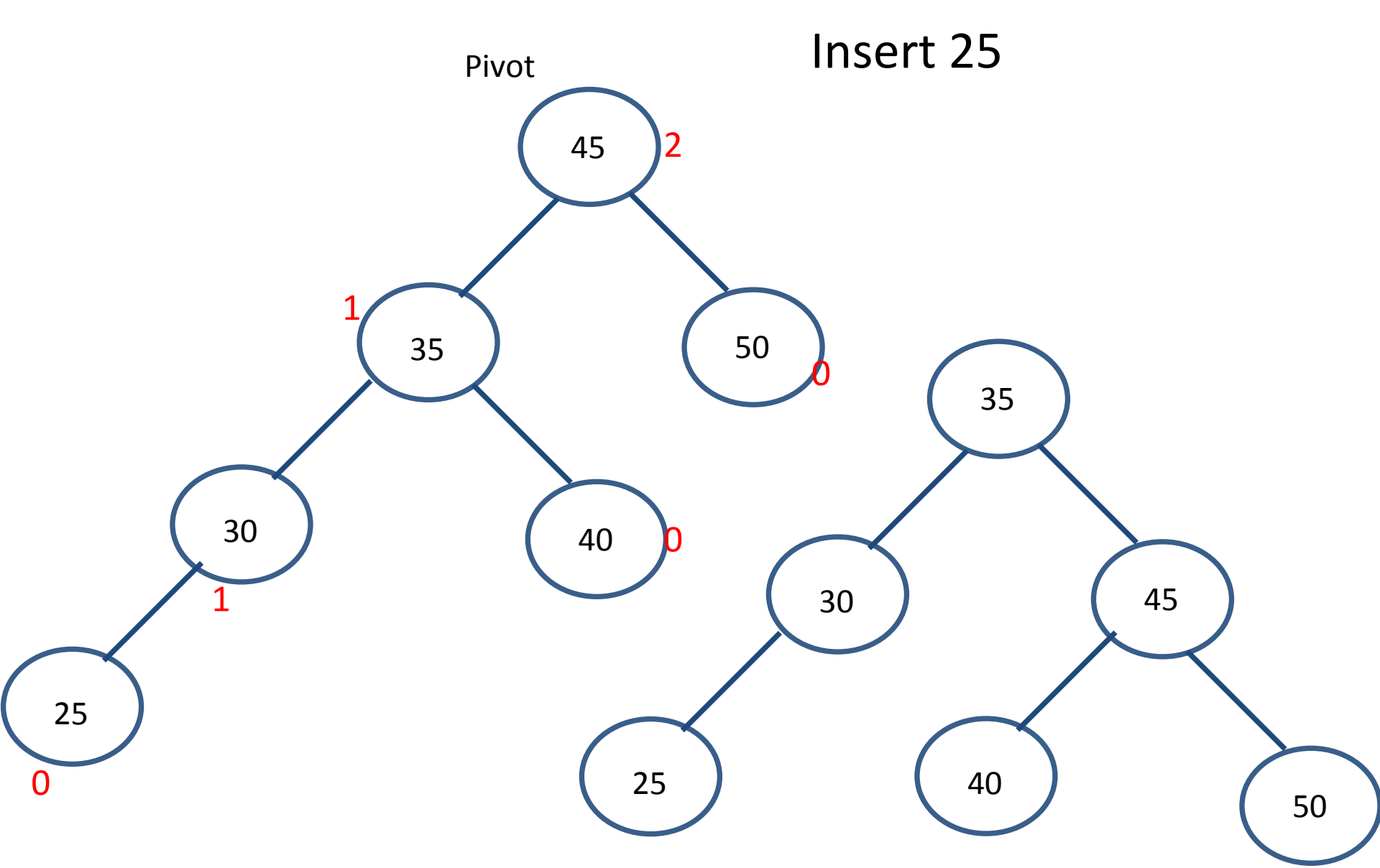
LL ROTATION



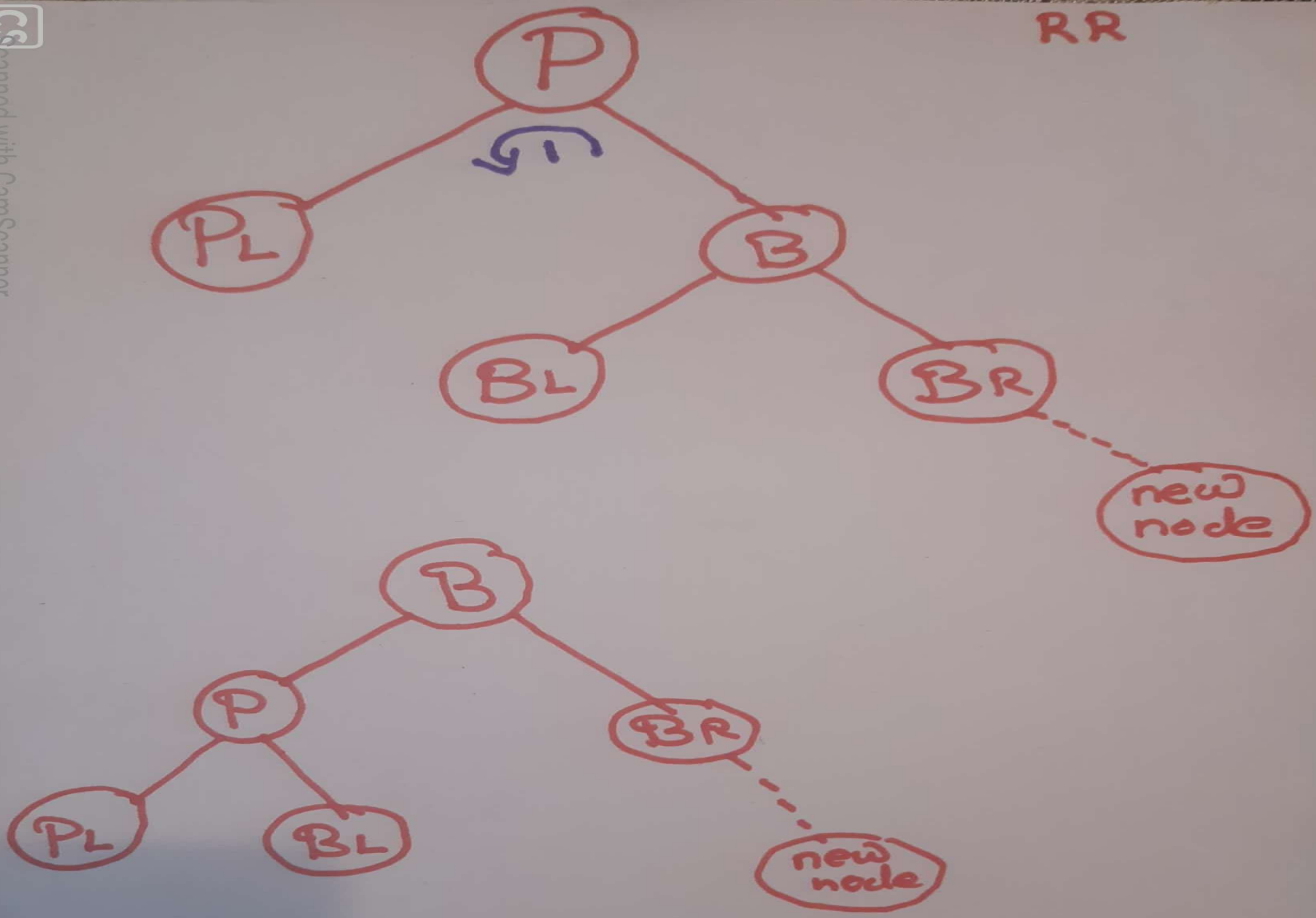
Insert 35



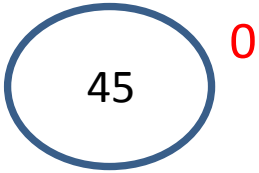

Insert 30
45
2
2
40
0
50
Pivot
1
35
30
0
LL
ROTATION
45
1
0
35
50
0
30
0
40
0


Insert 25
Pivot
45
2
1
35
50
0
30
40
0
1
25
0
35
30
45
25
40
50

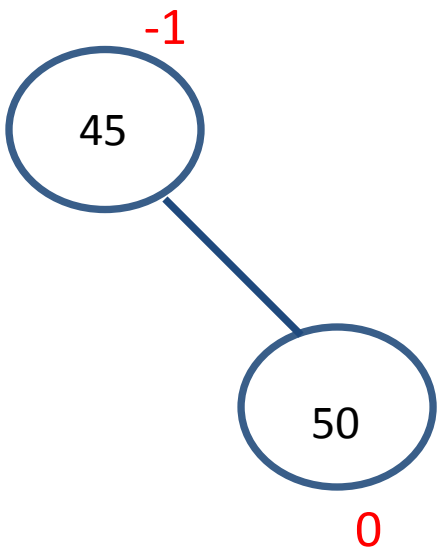
RR



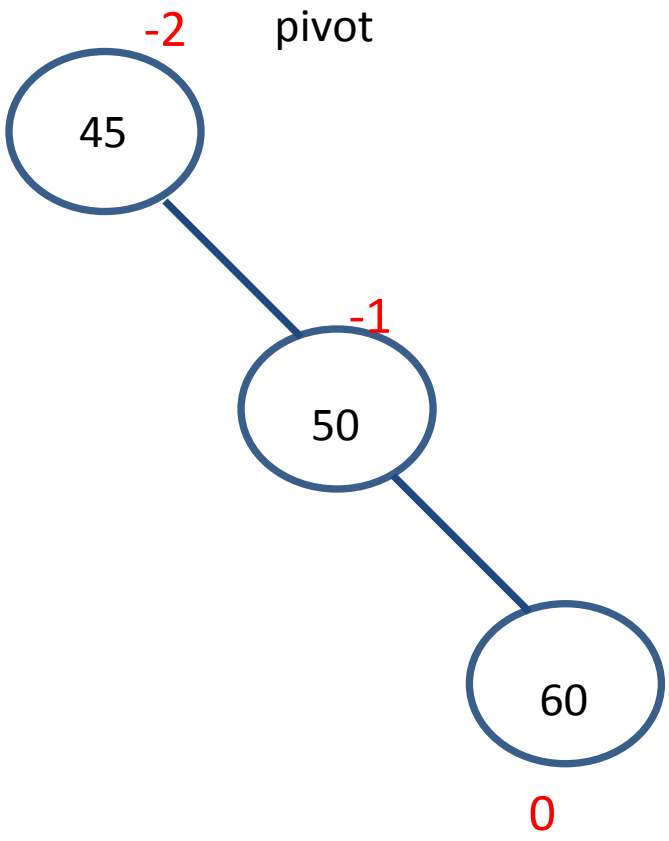
Insert 45

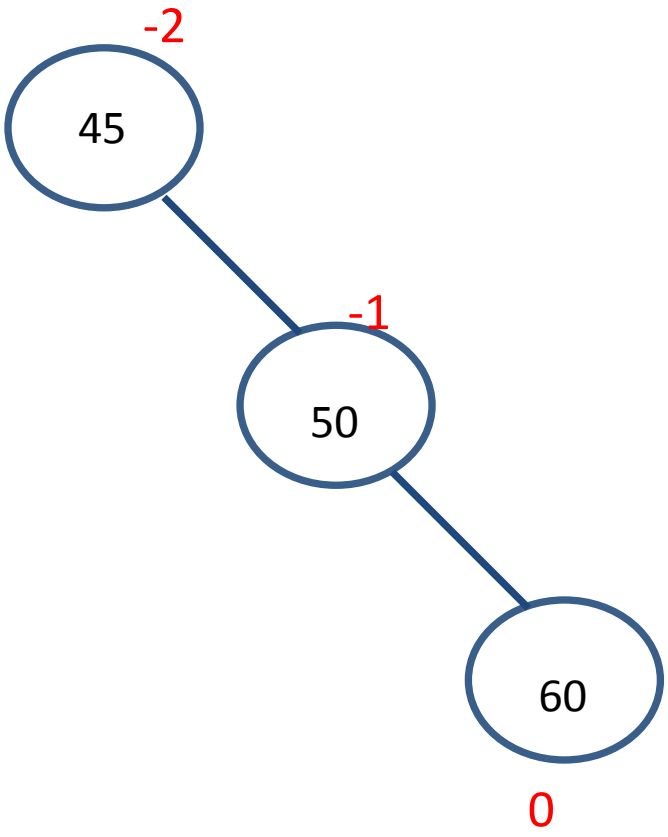


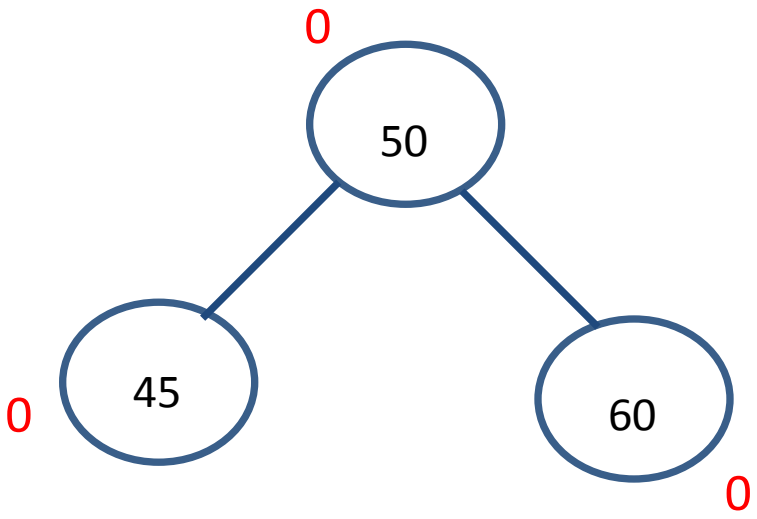
Insert 50



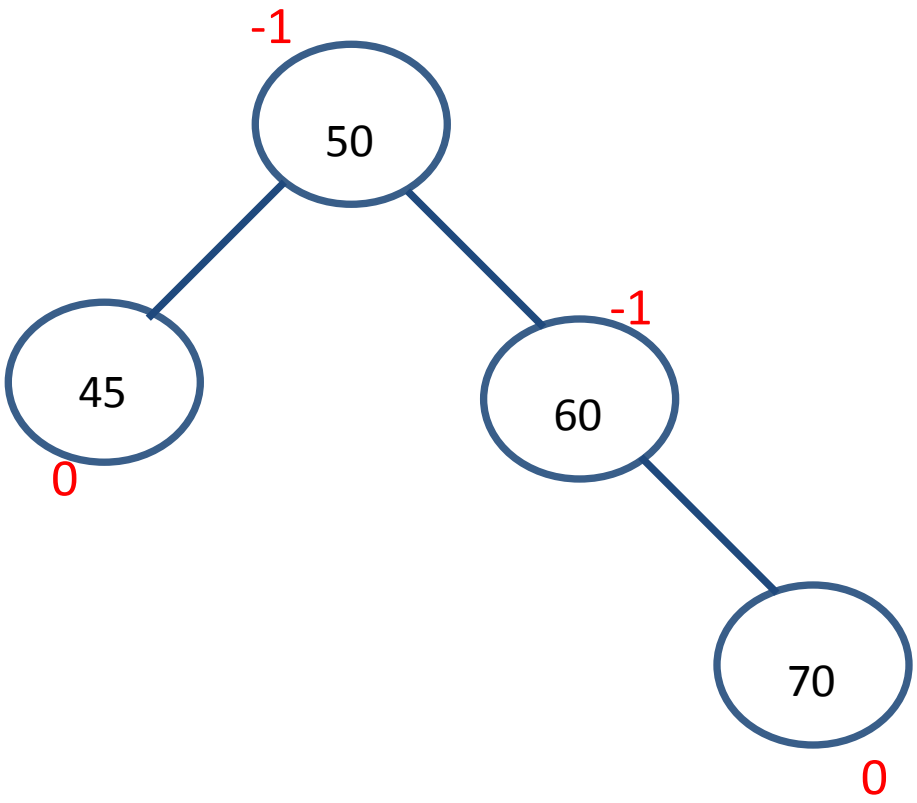
Insert 60

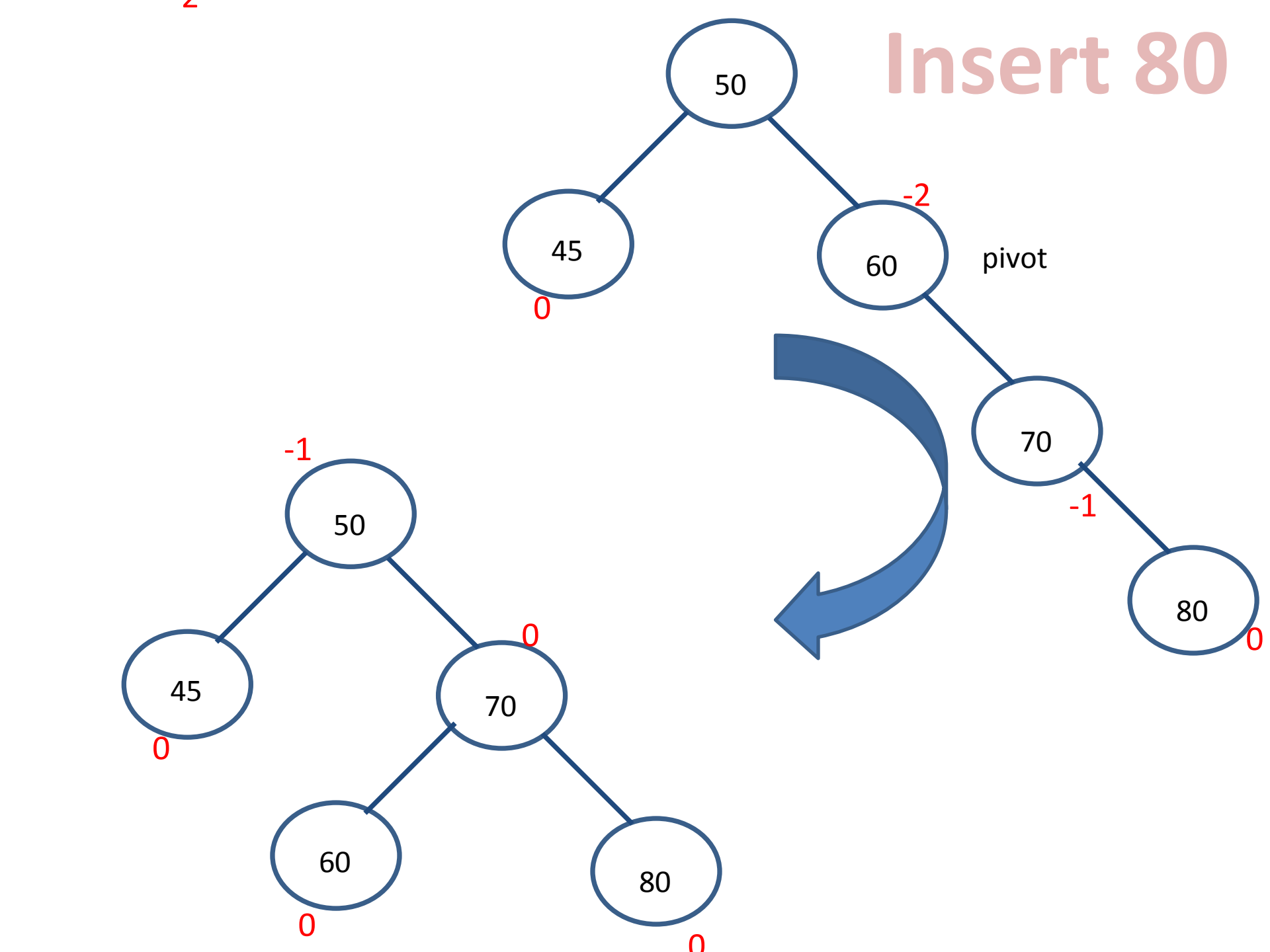



-2
45
-1
50
60
0

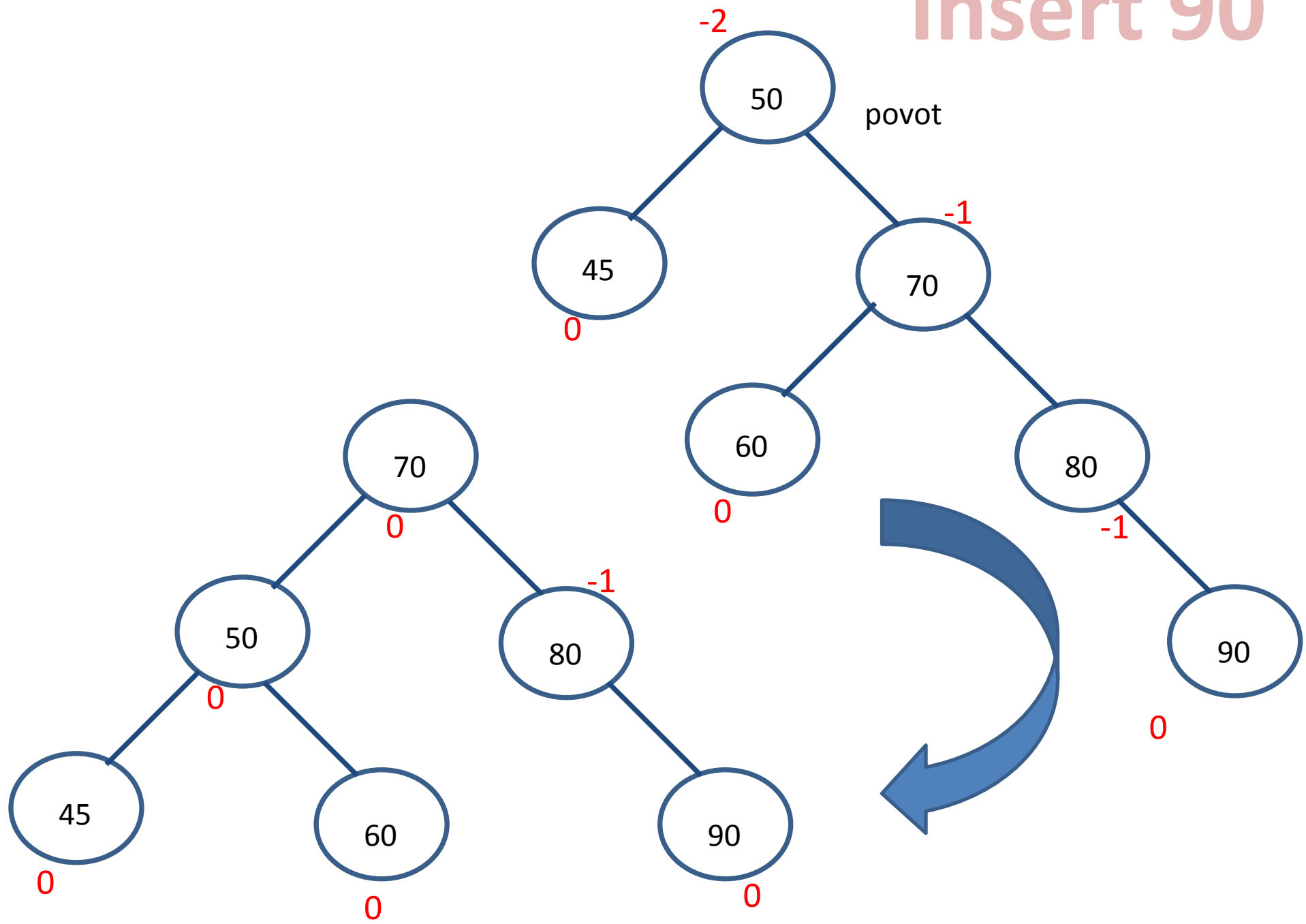

0
50
0
45
60
0

Insert 70

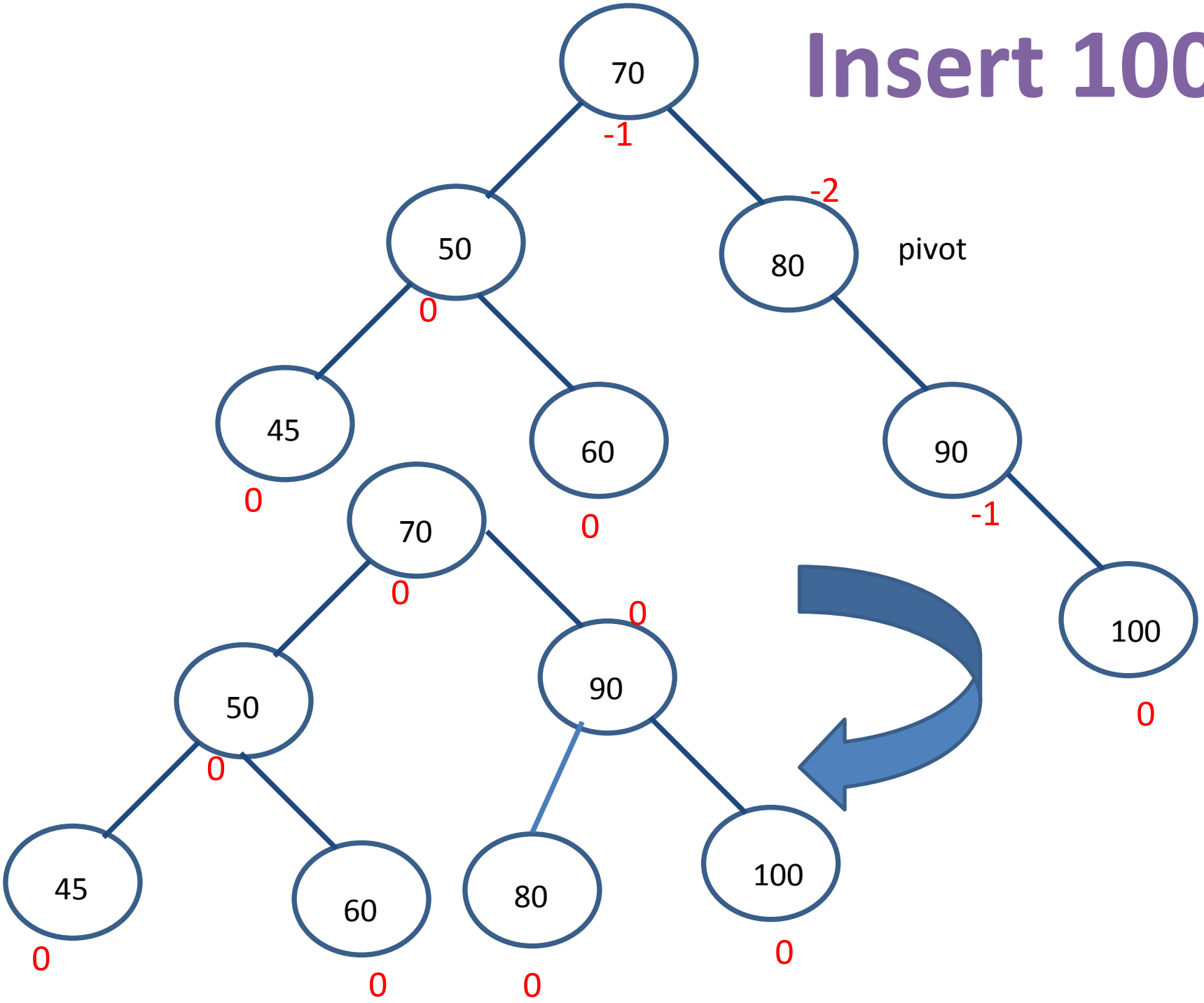



Insert 80
50
45
0
-2
60
pivot
70
-1
80
0
-1
50
45
0
0
70
60
0
80
0

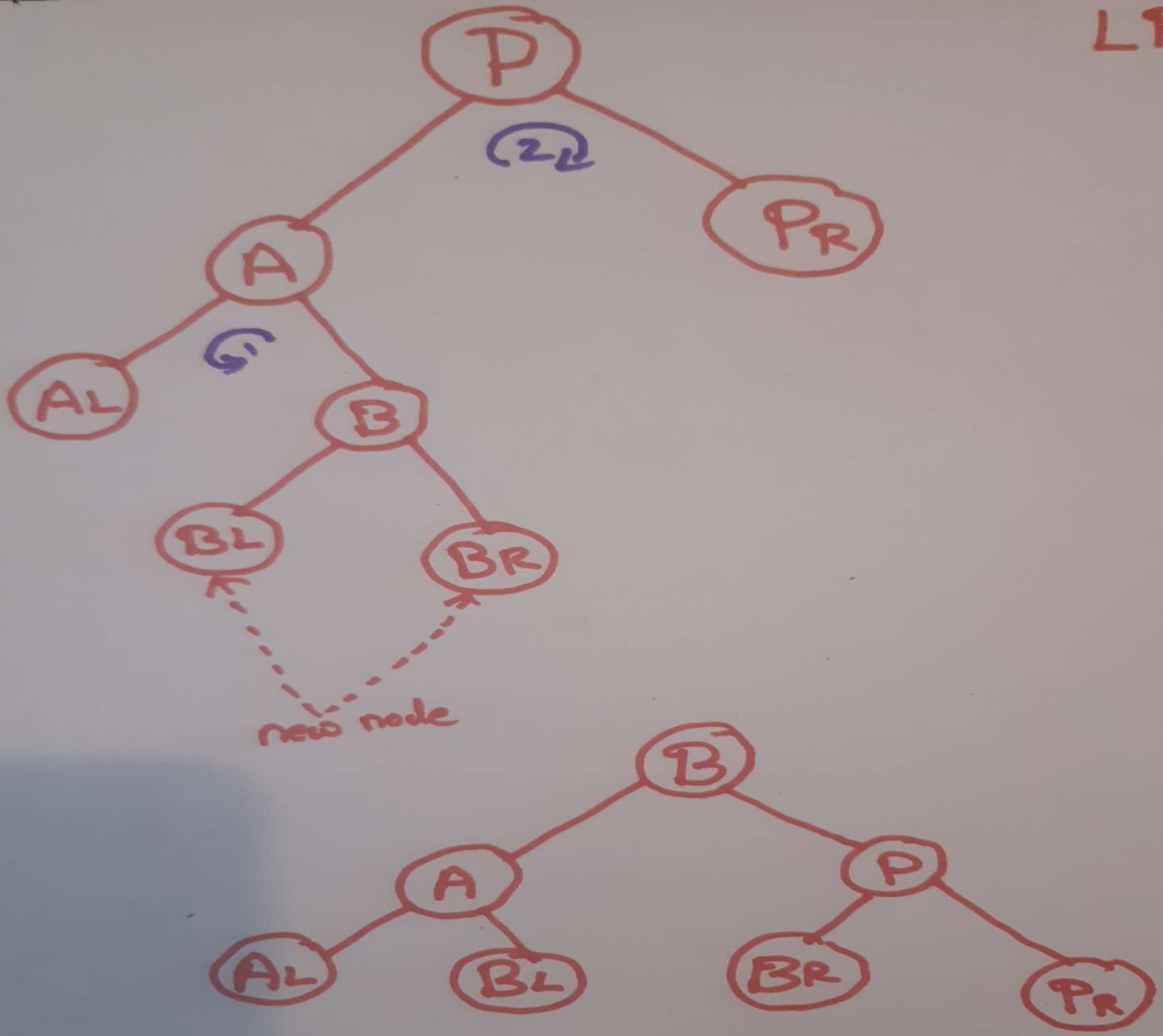
Insert 90



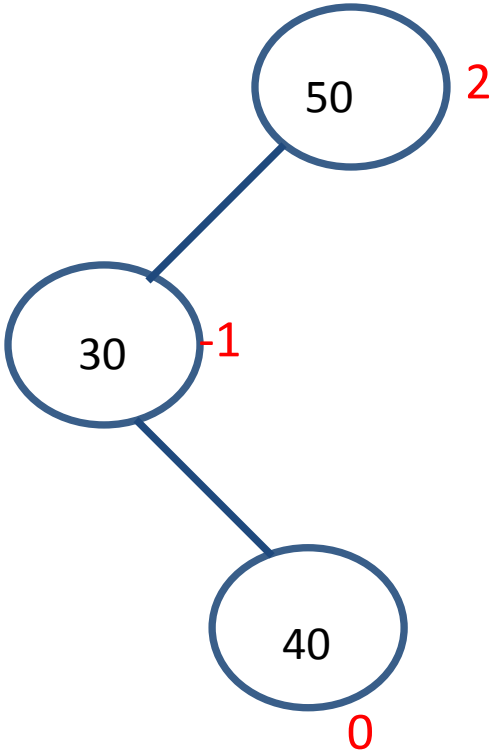
Insert 100


70
-1
50
-2
80
pivot
0
45
60
90
0
0
-1
100
0
70
0
0
50
90
0
0
45
60
80
100
0
0
0
0

LR

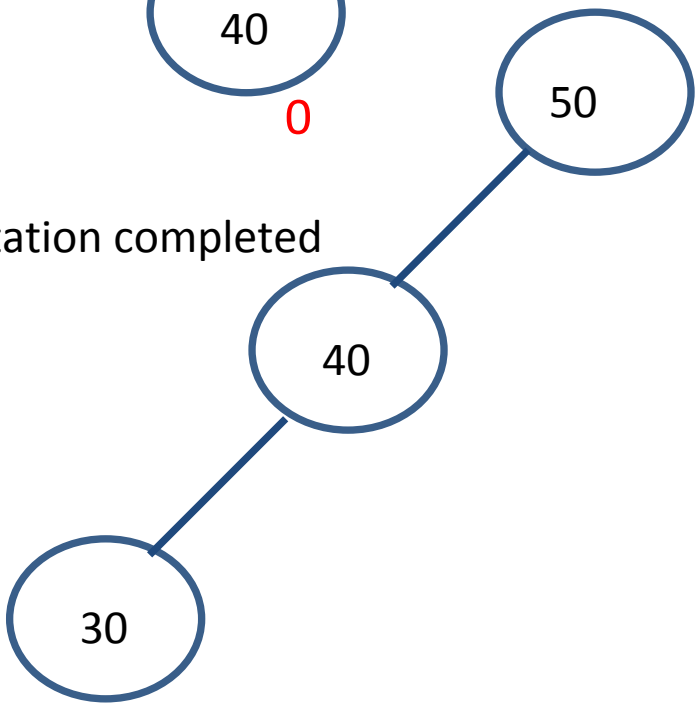


Insert 50,30,40

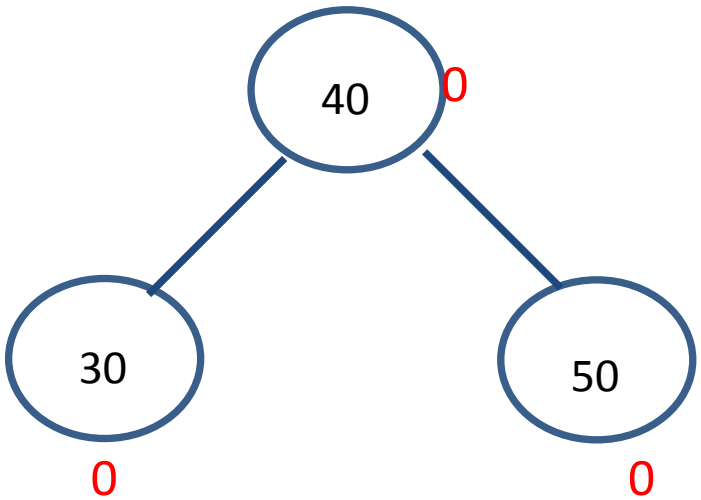


LR rotation

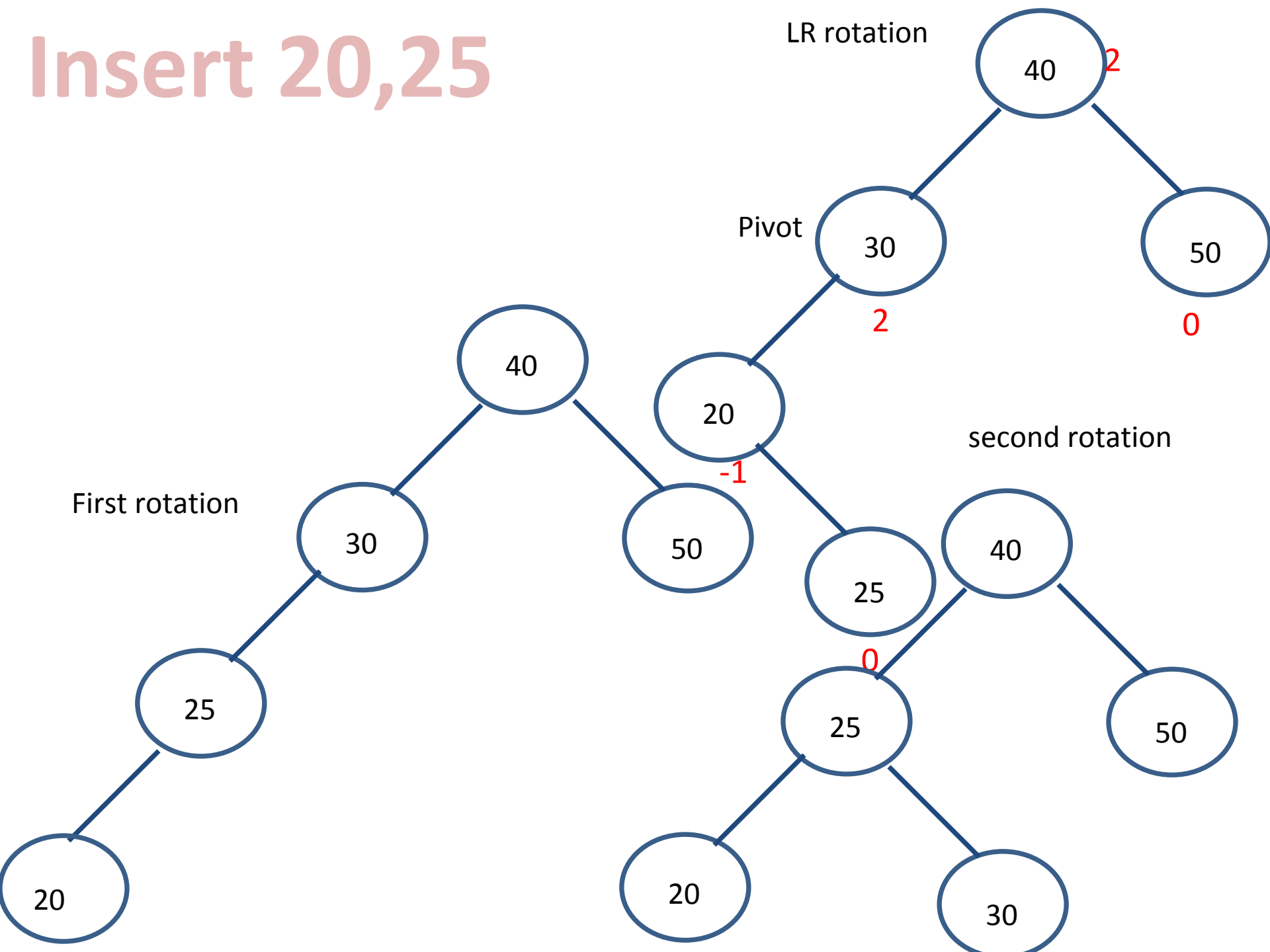
1st rotation completed



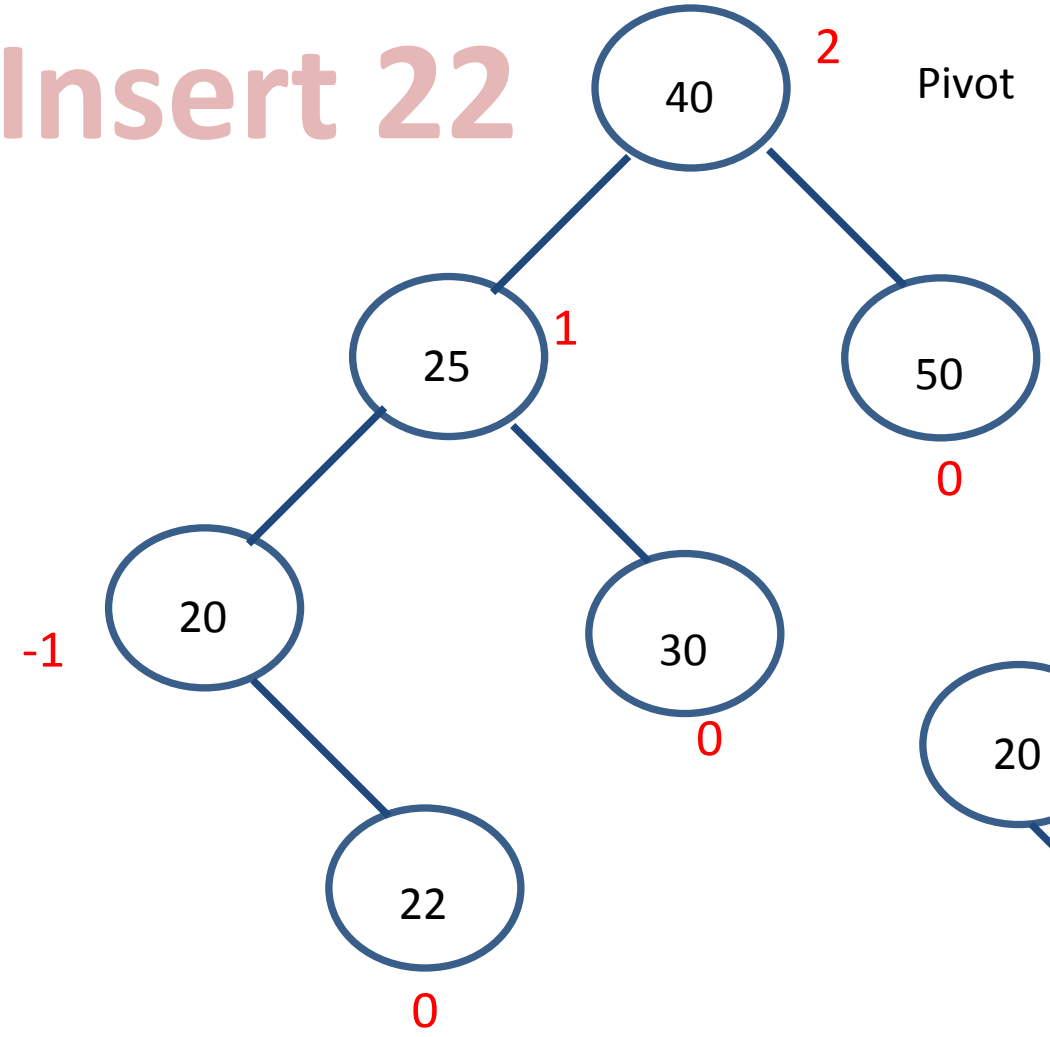
2nd rotation completed



Insert 20,25

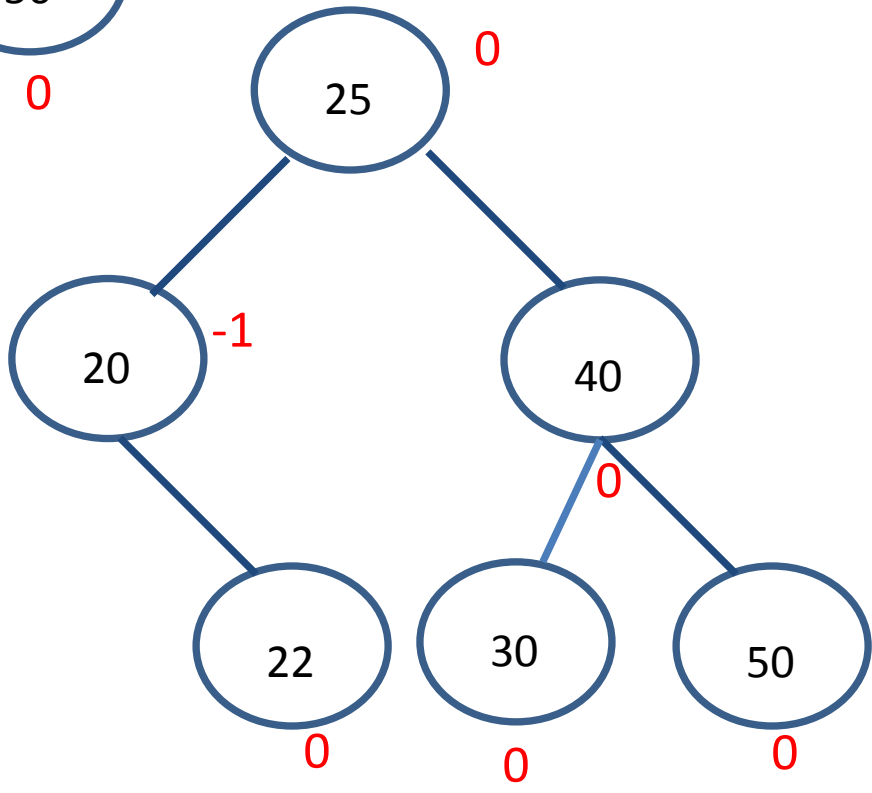

LR rotation
40
2
Pivot
30
50
2
0
40
20
-1
First rotation
second rotation
30
50
40
25
0
25
50
20
20
30

Insert 22

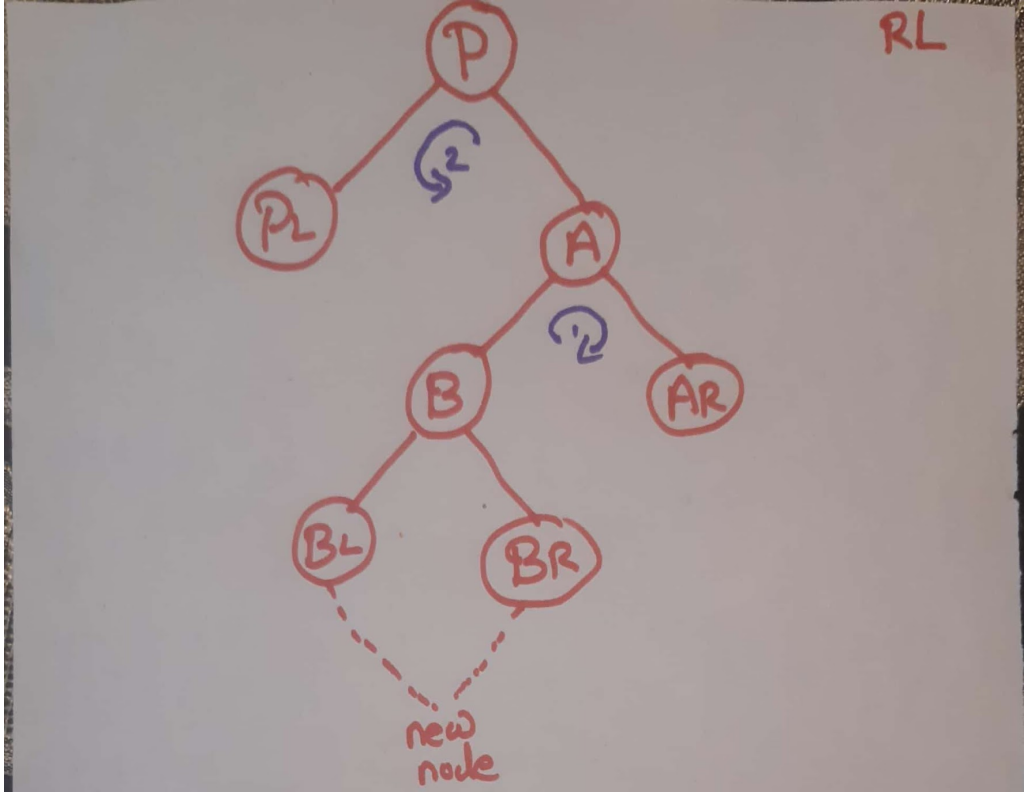


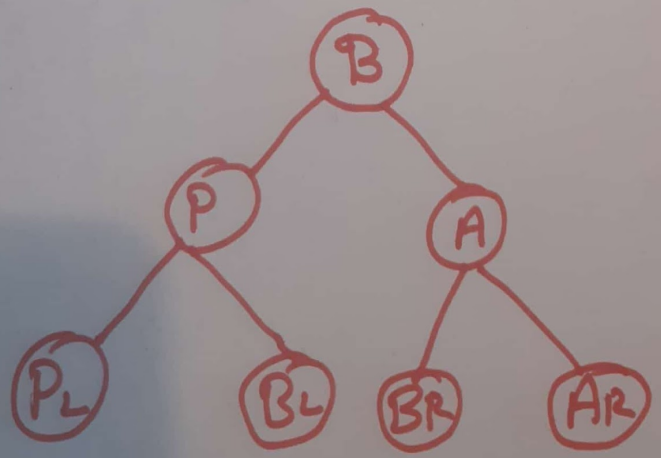
Which rotation?

LL rotation

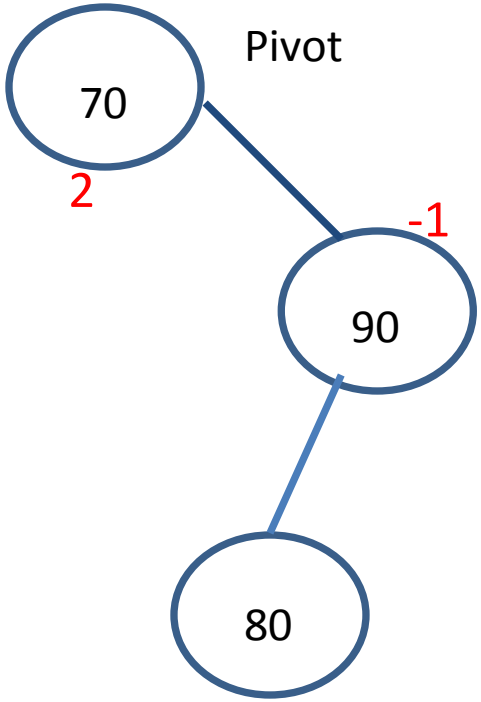


RL

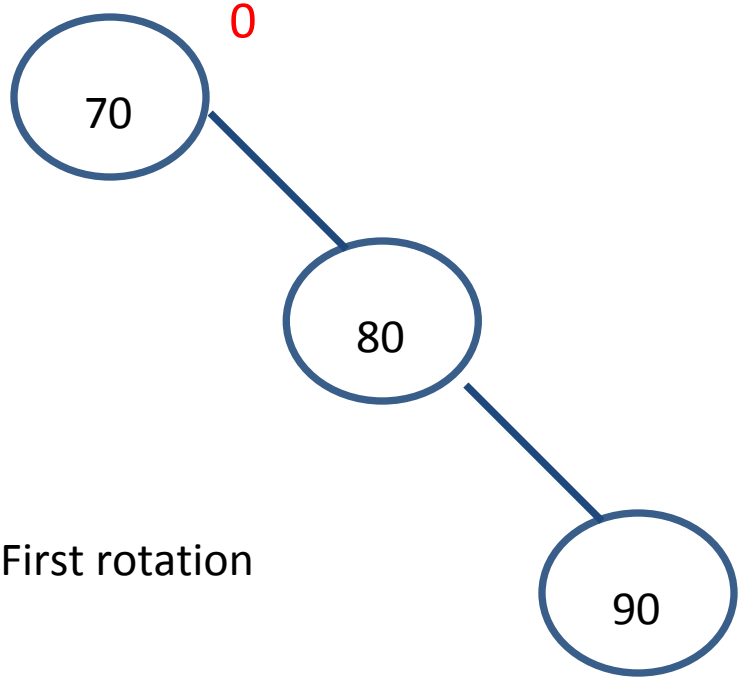




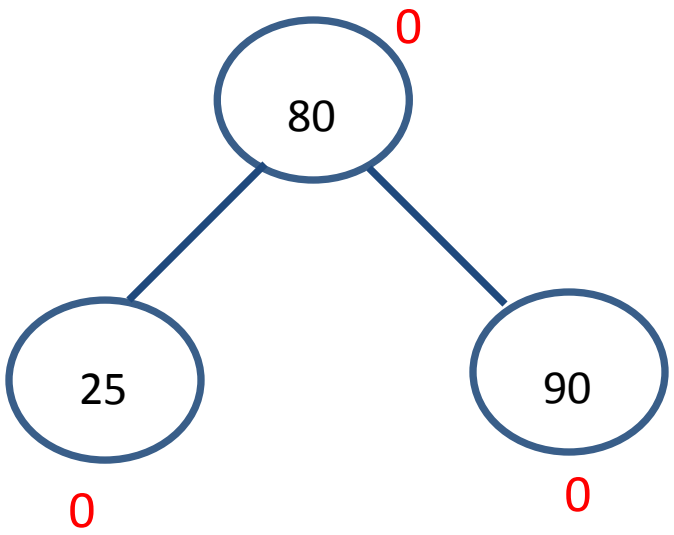
Insert 70, 90,80



RL rotation



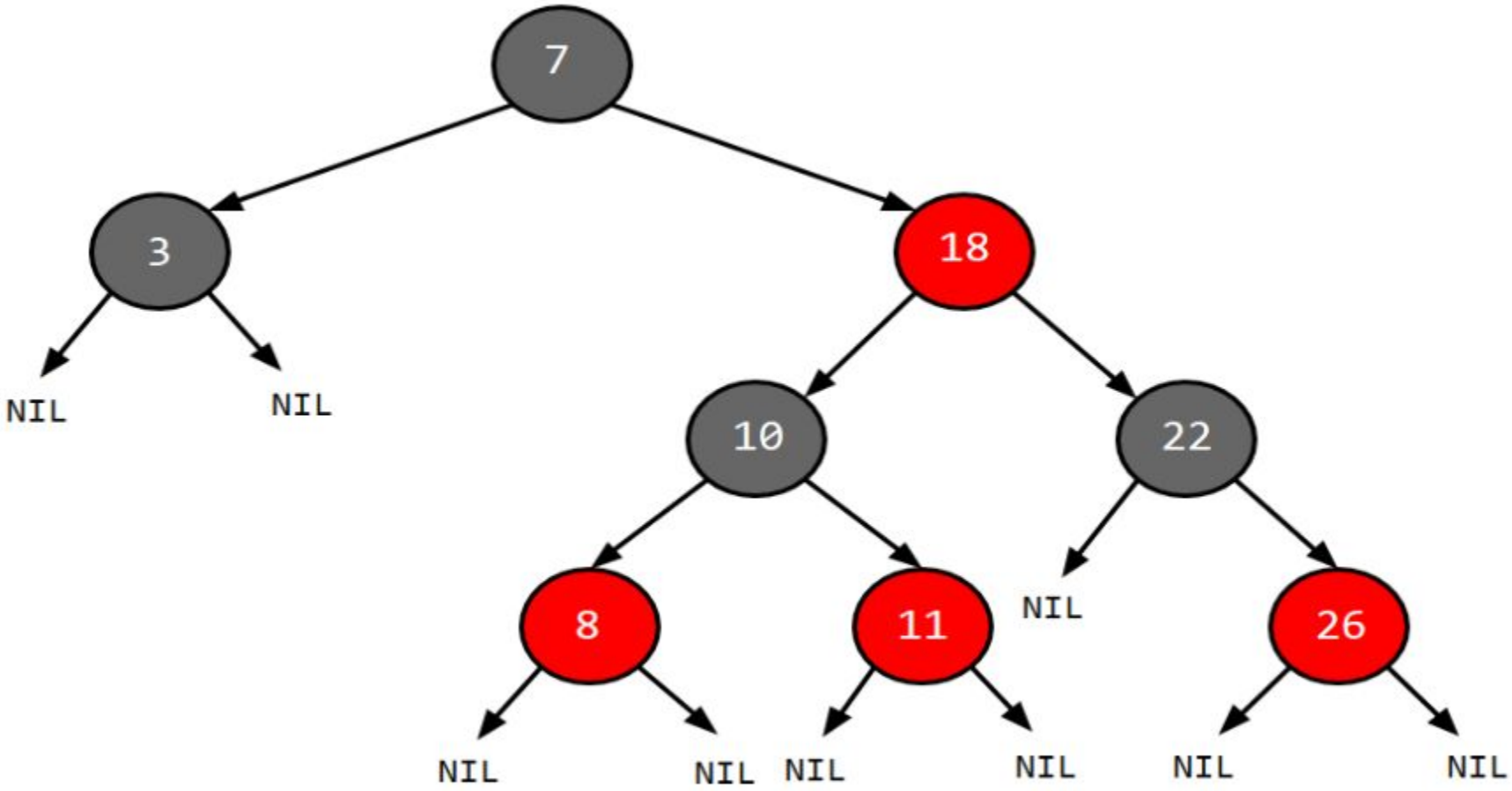
First rotation

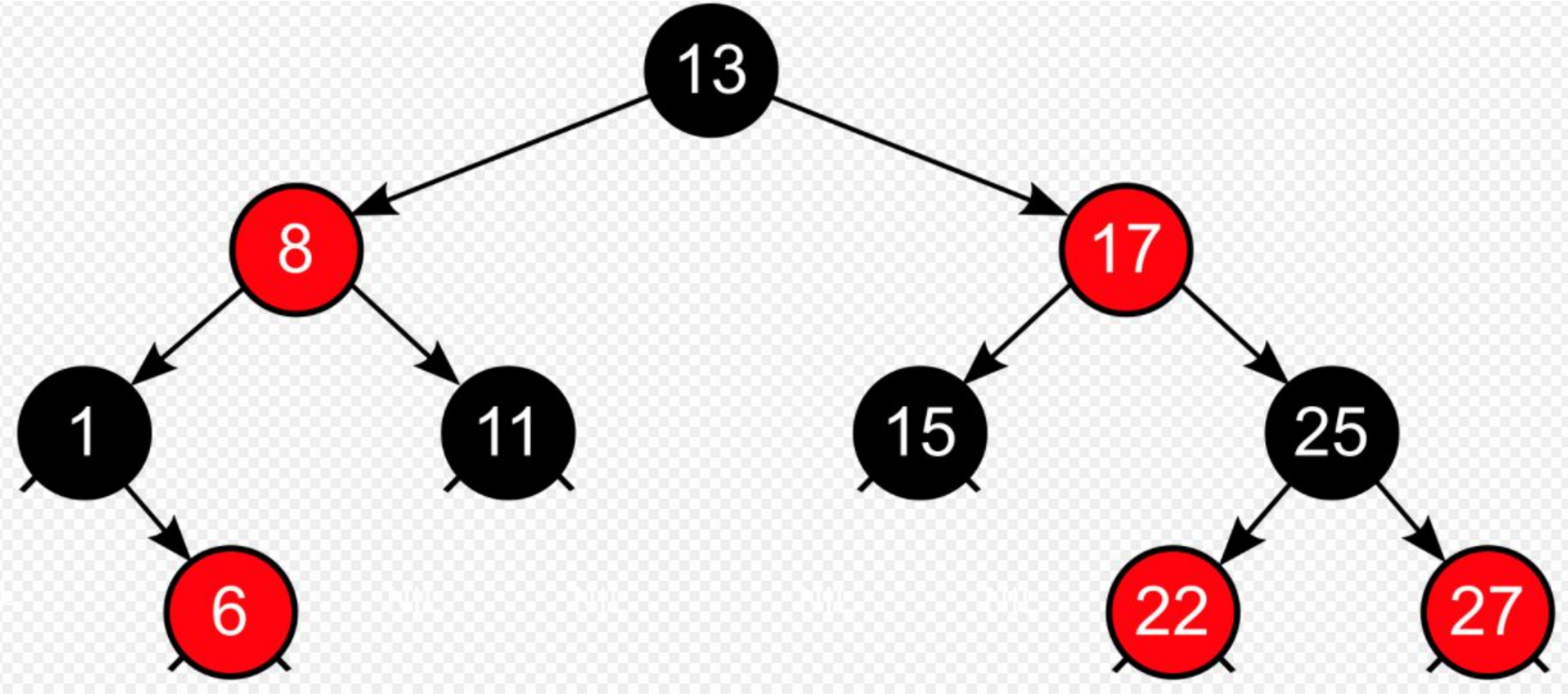


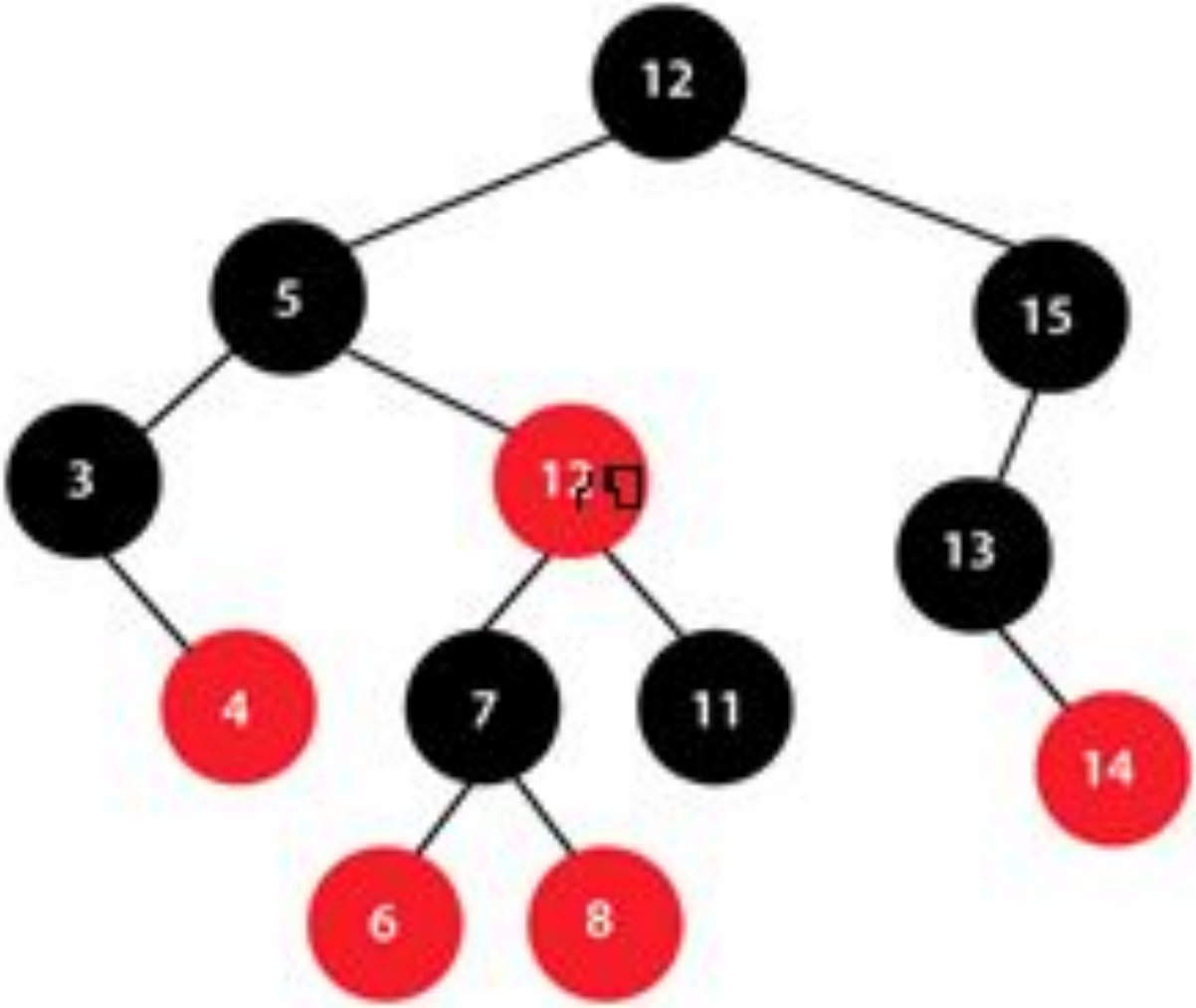
second rotation

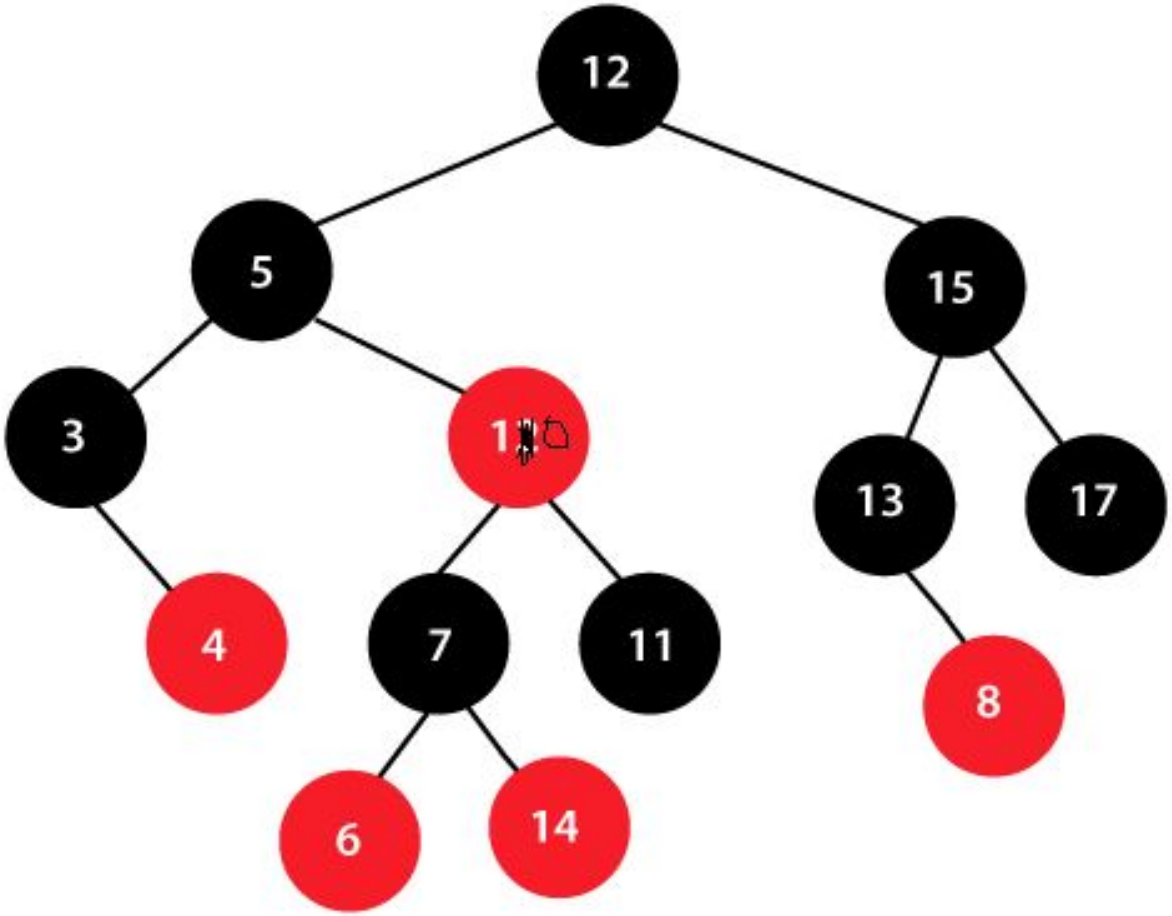
Red black tree

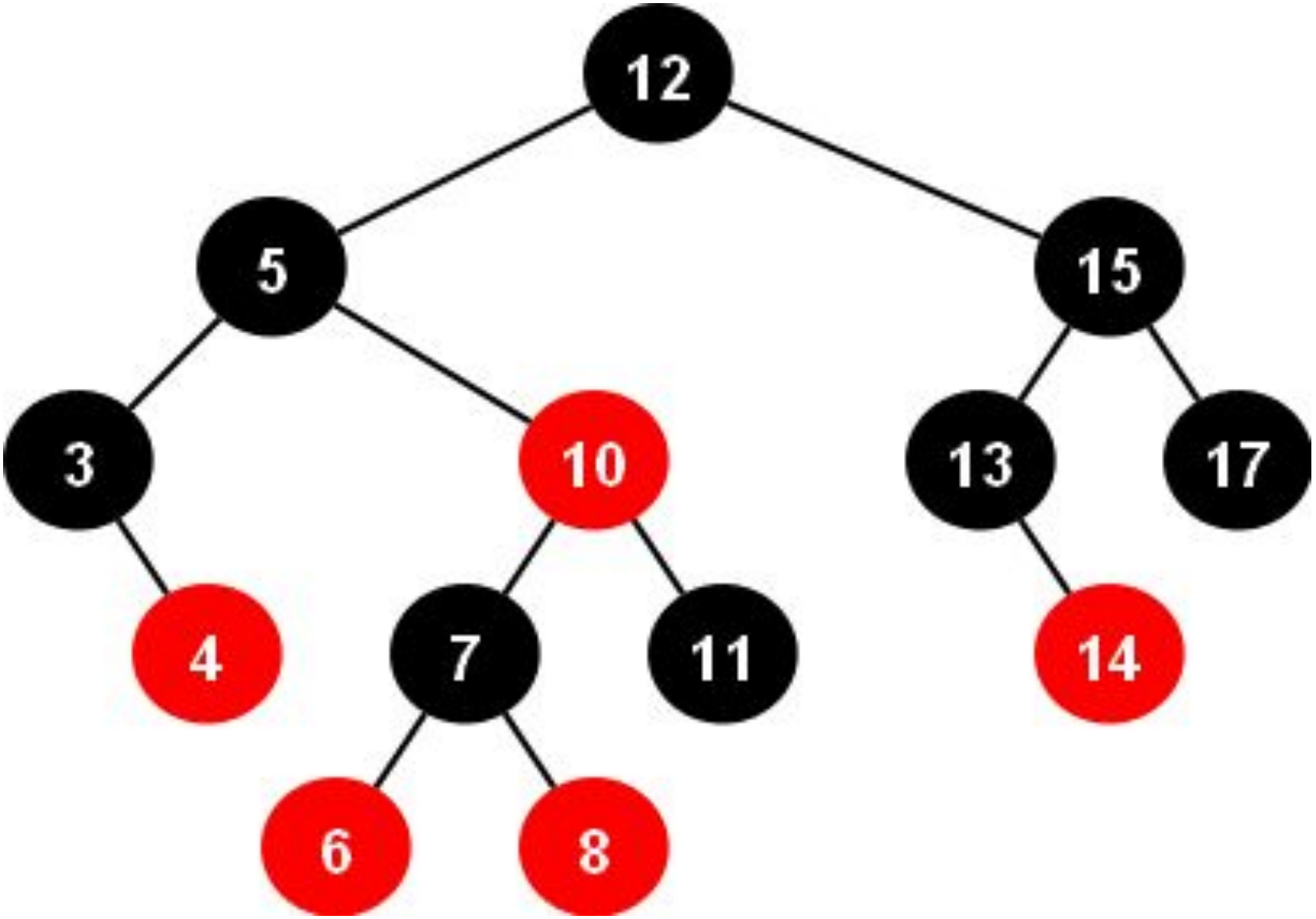
- It is a self balancing binary search tree
- Every node is either red or black
- Root is always black
- Every leaf, which is nil is black
- If a node is red, then its children are black
- Every path from a node to any of its descendent nil node has same no of black nodes
- The path from the root to furthest leaf node cannot be more than twice the length of the nearest shortest path of root.


7
3
18
NIL
NIL
10
22
8
11
NIL
26
NIL
NIL
NIL
NIL
NIL
NIL


13
8
17
1
11
15
25
6
22
27


12
5
15
3
13
4
7
11
14
6
8


12
5
15
3
13
17
4
7
11
8
6
14


12
5
15
3
10
13
17
4
7
11
14
6
8

Red black tree—Insertion

- If tree is empty, new node will be the root node with color black.
- If tree is not empty, new node will be red in color.
- If parent of new node is black, then exit
- If parent of new node is red, then check the color of the parent's sibling

  (a) if color is black or null, then do suitable rotation and recolor.

  (b) if color is red, then recolor and also check if parent's parent of new node is not root node, then recolor and recheck,

Red black tree—Deletion

Case 1:If node to be deleted is red, simply delete it.

Case 2: if root is double black, just remove DB

Case 3: if DB's sibling is black and both of its children are black

(i)Remove DB

(ii) Add black to its parent (if P is red, it becomes black. If P is black, it becomes double black)

(iii) make sibling red

(iv) if DB exists, apply other cases

Case 4: if DB's sibling is red and both of its children are black

(i) swap colors of parent and sibling

(ii) rotate parent in DB's direction

(iii) if DB exists, apply other cases

Case 5: if DB's sibling is black, sibling's child who is far from DB is black, near child to DB is red

(i) swap color of DB's sibling and sibling's child who is near to DB

(ii) Rotate sibling in opposite direction to DB

(iii) Apply case 6

Case 6: DB's sibling is black, far child is red

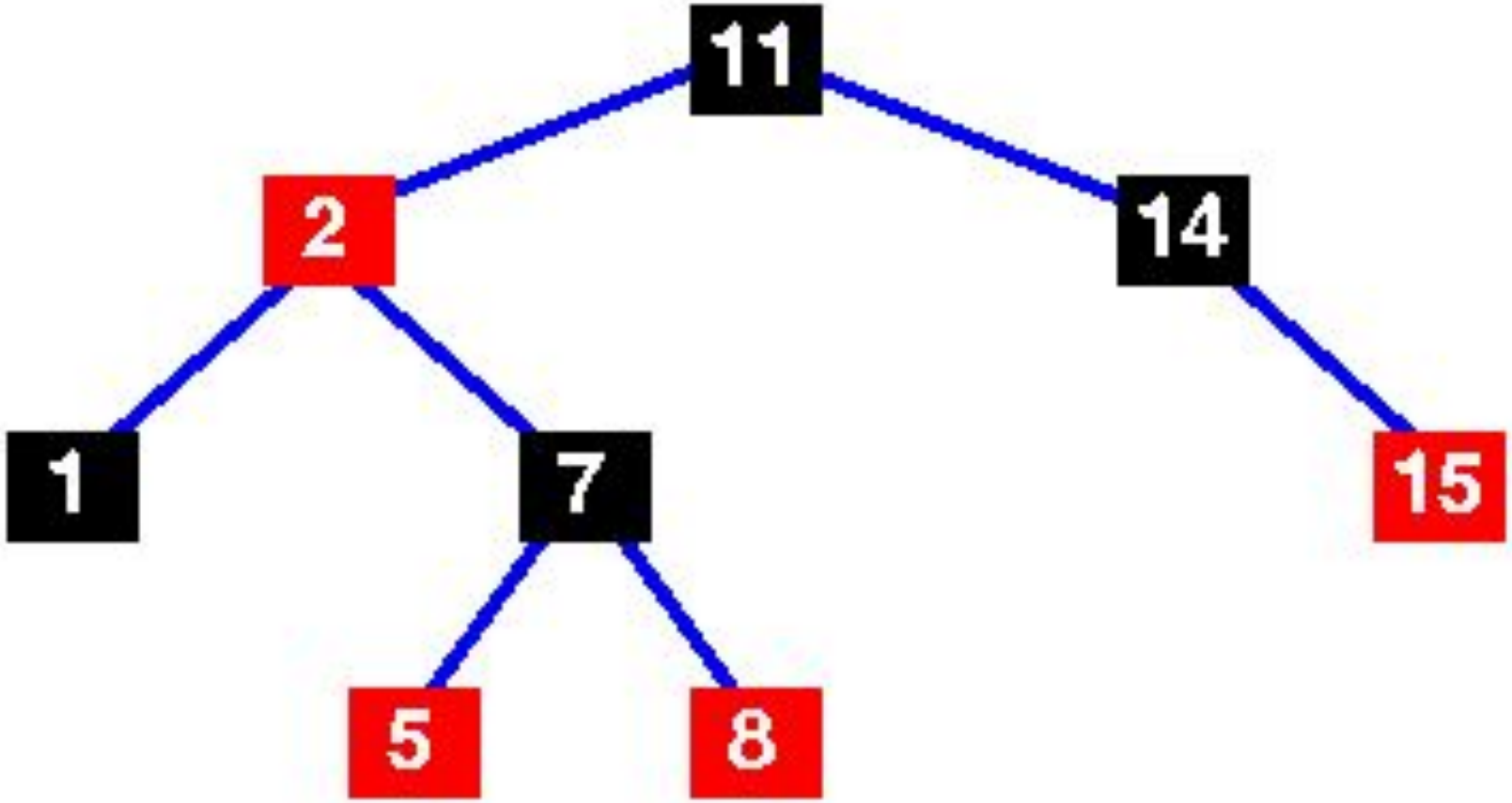
(i) swap color of parent and sibling

(ii) Rotate parent in DB's direction

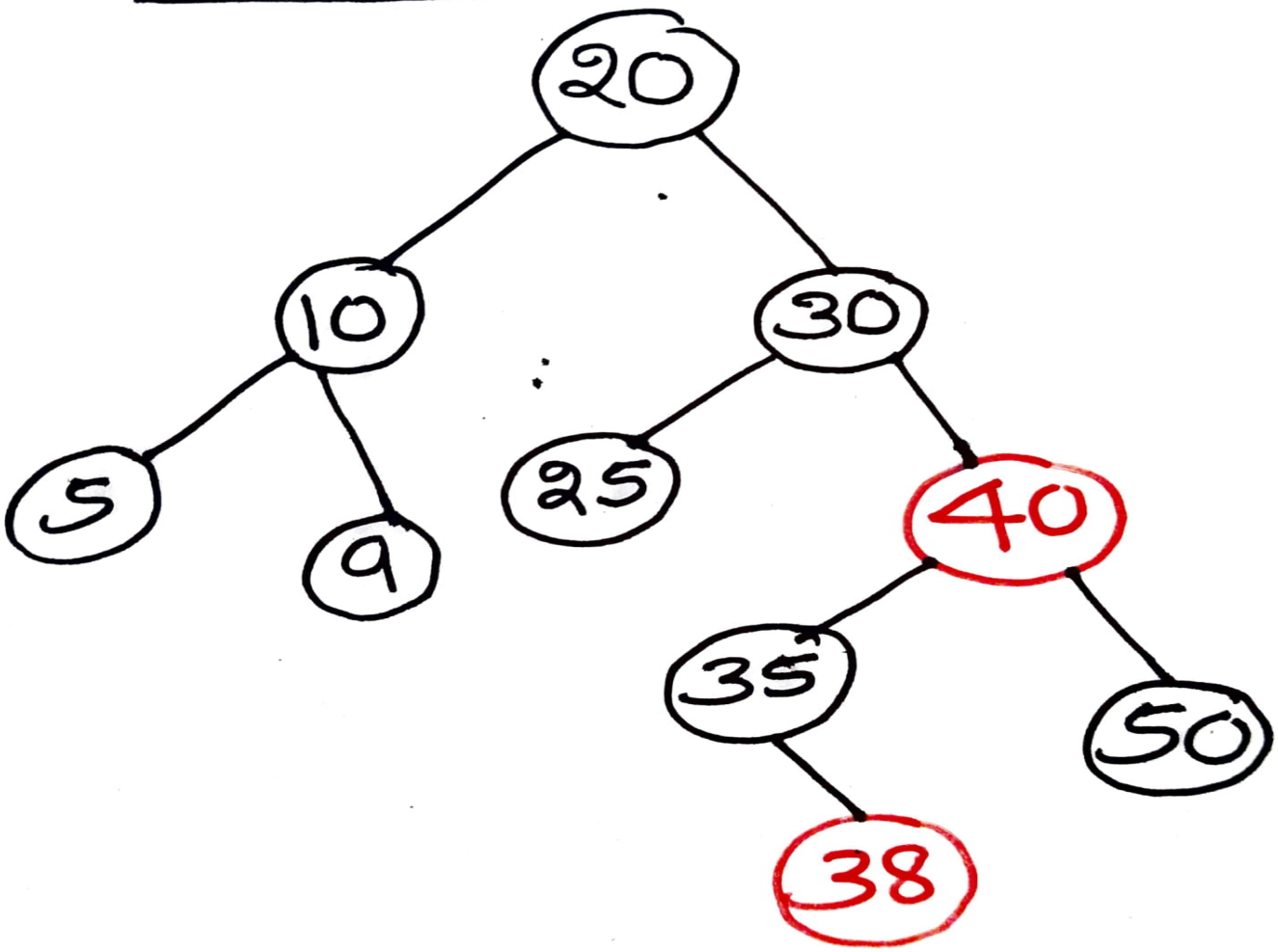
(iii) Remove DB

(iv) change color of red child to black

Case 1: delete 5 or 8 or 15 or 2 or 14 or 7



Delete 30



B-Tree

- Also known as Balanced M way tree (m is the order or degree)
- Can have more than 1 key, and more than 2 children
- Maintain data in sorted order
- All leaf node must be in same level.
- B tree of order M has the following properties:
  - Every node has Max M children
  -Minimum children
    ----Root—2(if there is any child)
    ----Leaf—0
    ----internal nodes---M/2

- Every node has maximum m-1 keys
- Min keys

  ----Root node-1

  ----All other nodes (M/2) -1
- Insertion will only be in the leaf node

Splay Tree

Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.

In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called **"Splaying"**.

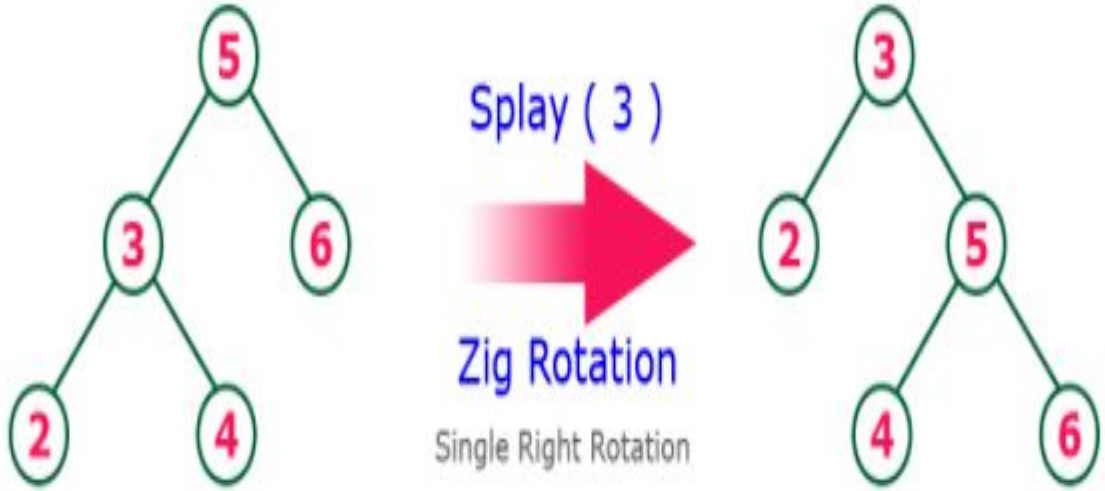
Splaying an element, is the process of bringing it to the root position by performing suitable rotation operations.

Rotations in Splay Tree

- 1. Zig Rotation
- 2. Zag Rotation
- 3. Zig - Zig Rotation
- 4. Zag - Zag Rotation
- 5. Zig - Zag Rotation
- 6. Zag - Zig Rotation

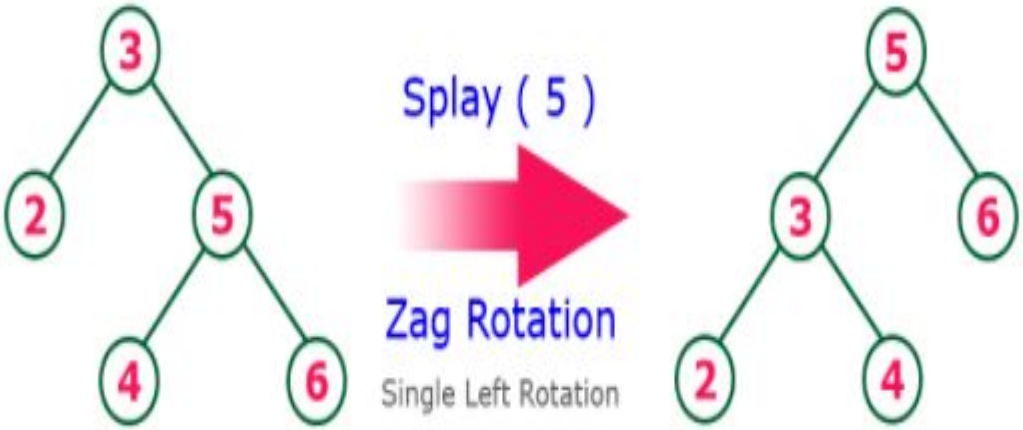
Zig Rotation

The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...



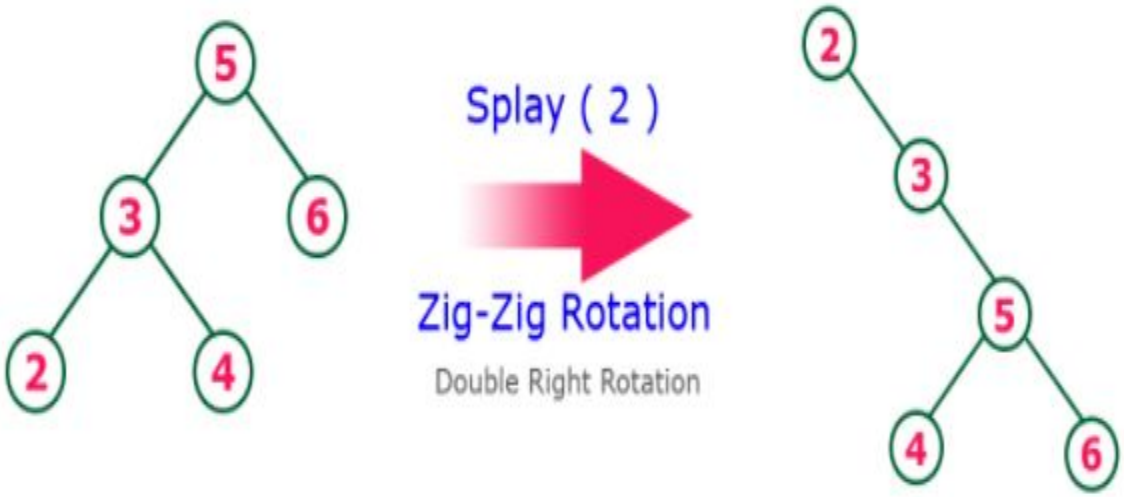
Zag Rotation

The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...



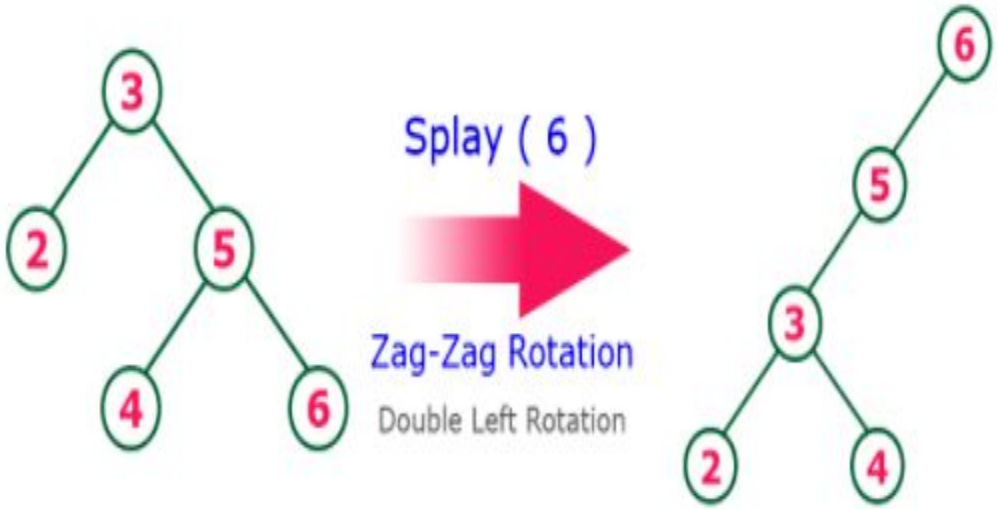
Zig-Zig Rotation

The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...



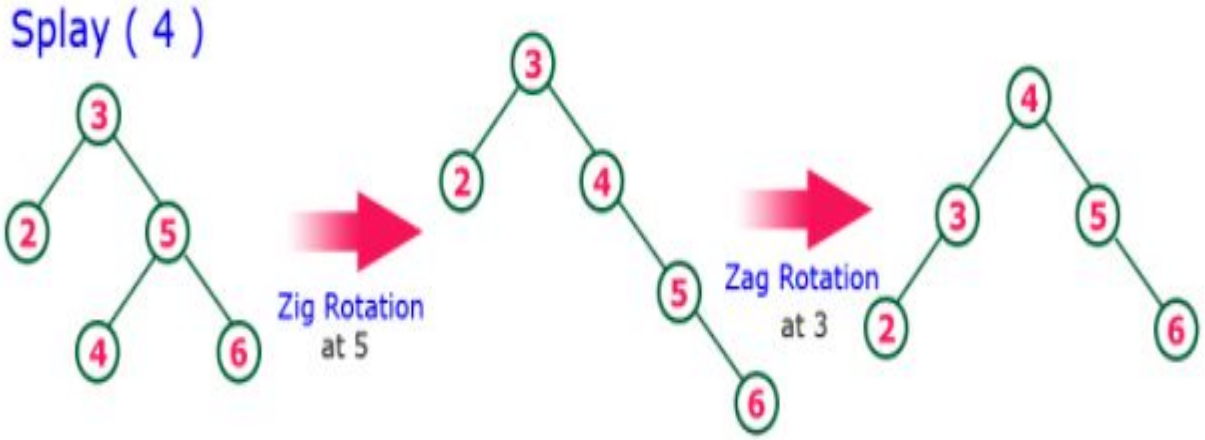
Zag-Zag Rotation

The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...



Zig-Zag Rotation

The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



Zag-Zig Rotation

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...