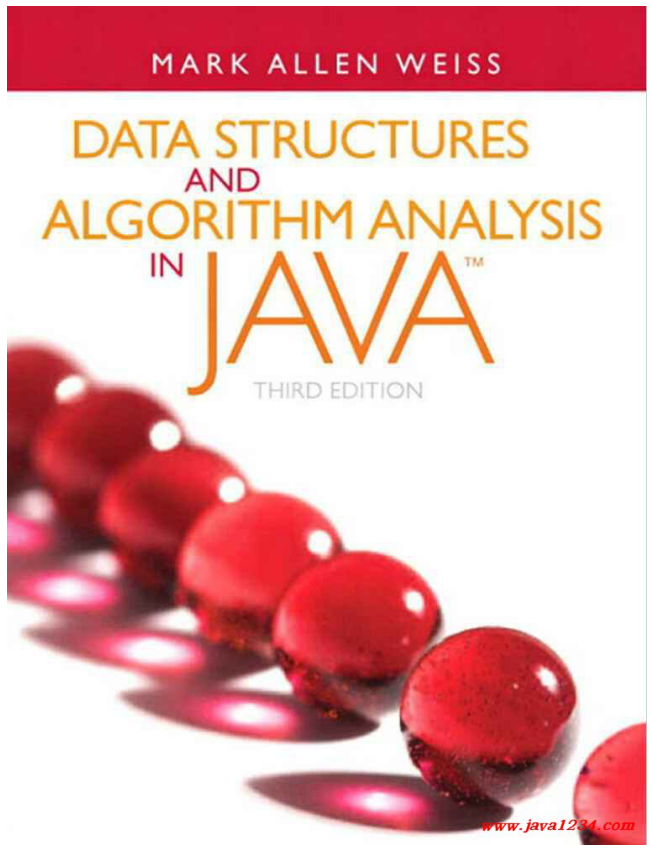
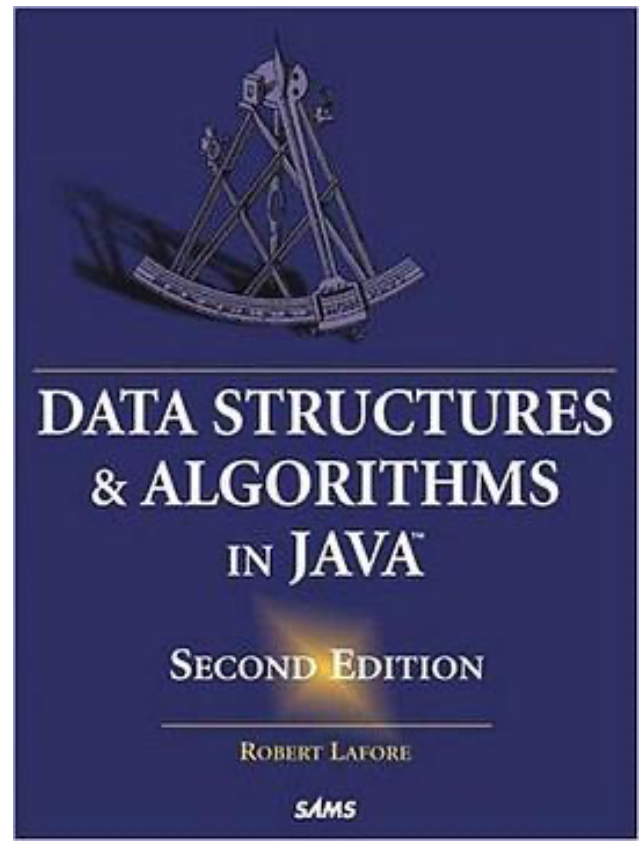
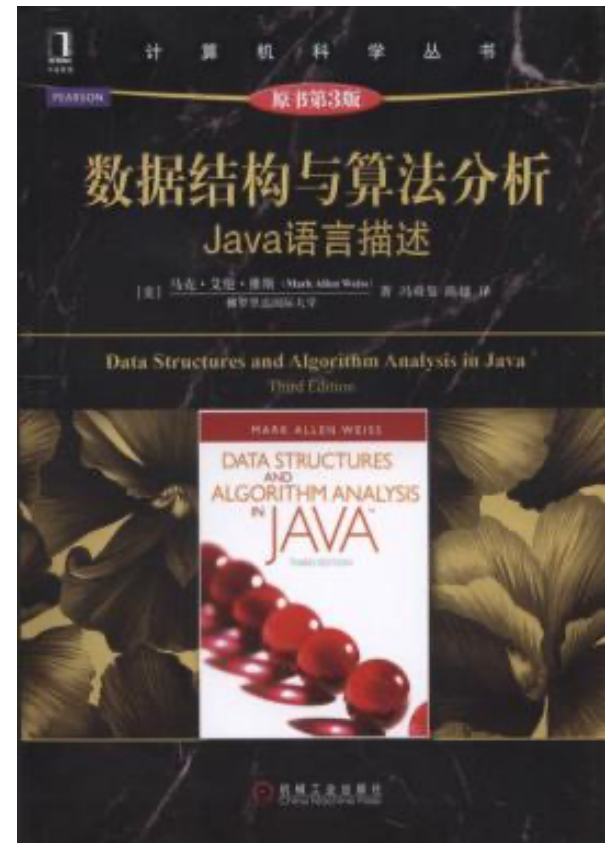
Topic 13 – AVL Tree

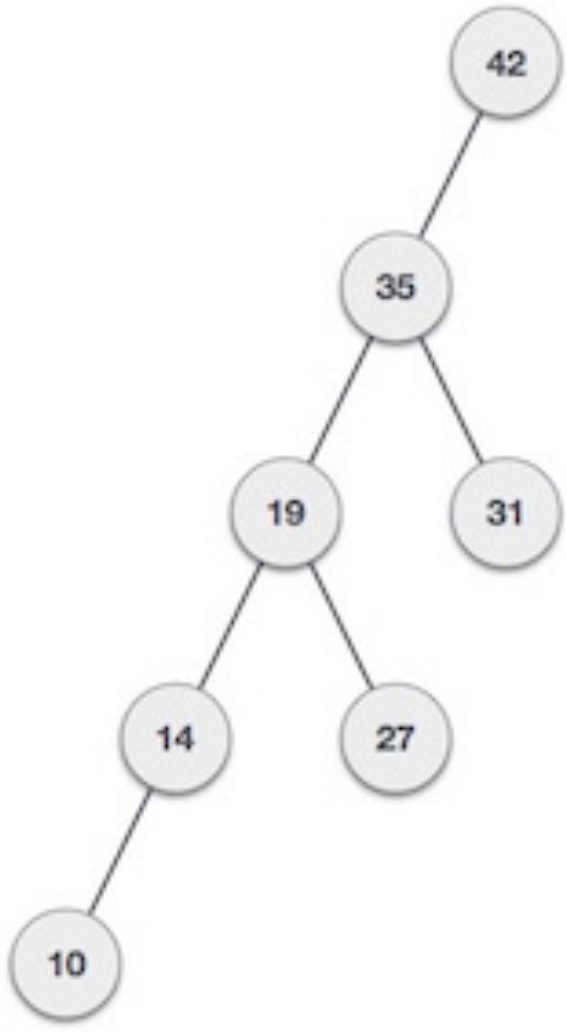




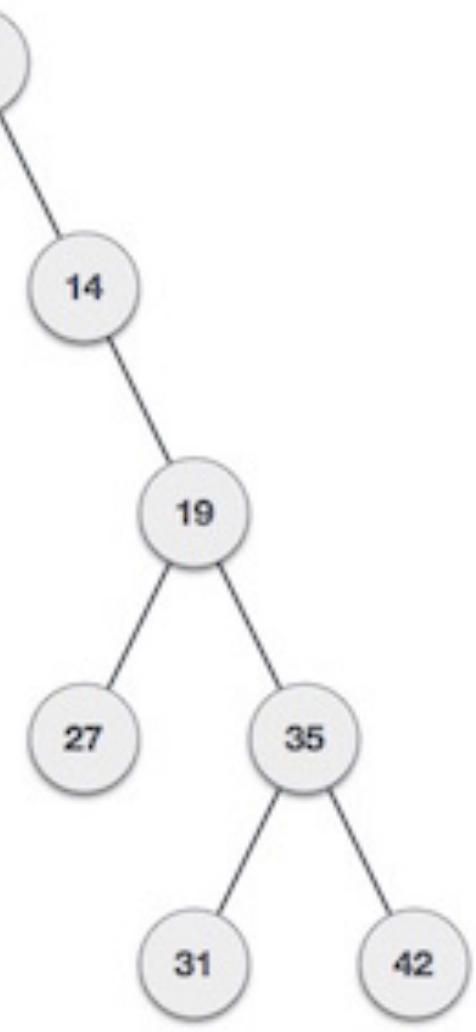


Motivation

- What if the input to binary search tree comes in a sorted (ascending or descending) manner?
- It will then look like this –



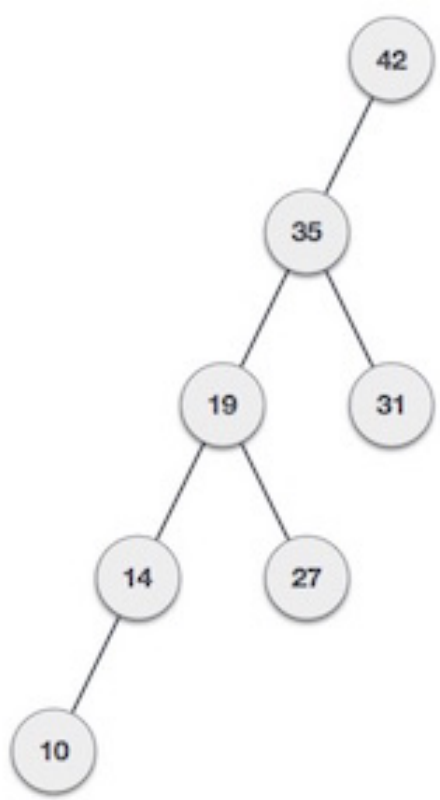
If input 'appears' non-increasing manner



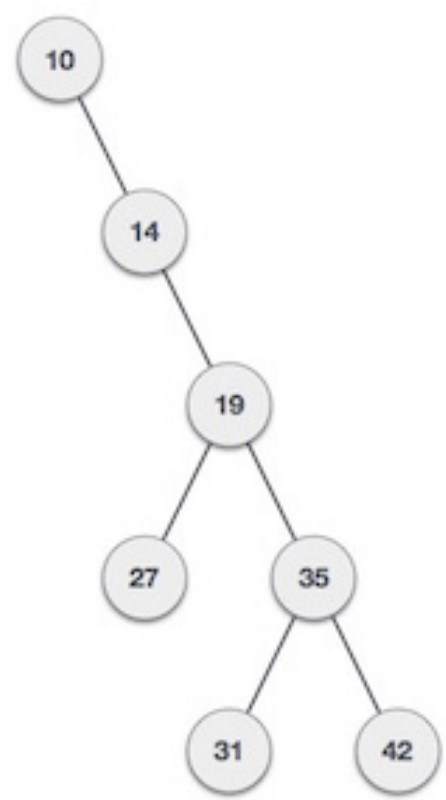
If input 'appears' in non-decreasing manner

Motivation

- It is observed that BST's worst-case performance is closest to linear search algorithms, that is **O(n)**.
- In real-time data, we cannot predict data pattern and their frequencies.
- So, a need arises **to balance** out the existing BST.



If input 'appears' non-increasing manner

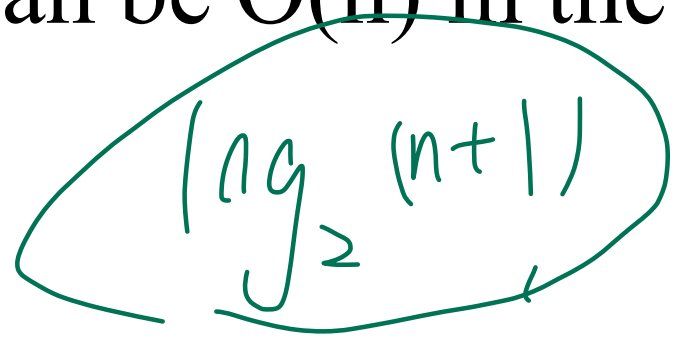


If input 'appears' in non-decreasing manner

- **AVL Tree**
- Rotation
- Insert
- Delete
- Max and Min
- Driver code
- Complexities of Different Operations

Balanced Binary Tree

- The disadvantage of a binary search tree with n nodes is that its height can be as large as n-1.
- This means that the time needed to perform insertion and deletion and many other operations can be O(n) in the worst case.
- We want a tree with small height.
- A binary tree with *N* node has height at least Θ(log *n*)
- Thus, our goal is to keep the height of a binary search tree *O*(*log n*).
- Such trees are called *balanced binary search trees*.
  - Examples are AVL tree, red-black tree.

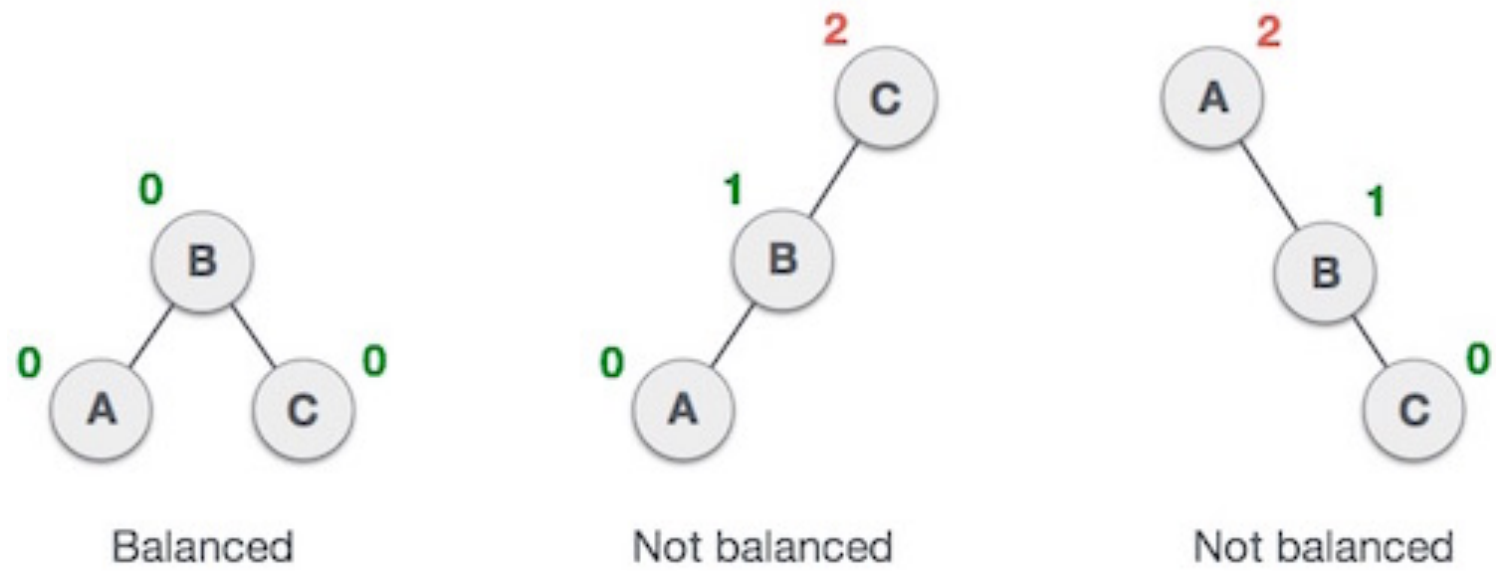


AVL Tree

- **AVL trees** are ***height balancing binary search tree***.
  - Georgy Adelson-Velsky and Evgenii Landis propose it at 1962.
- AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1.
  - -1, 0, 1
  - Note. Height of a node
    - The height of a leaf is 1.
    - The height of an internal node is the maximum height of its children plus 1.
- This difference is called the **Balance Factor**.
  - height of the left subtree - height of the right subtree
  - We hope … **Balance Factor** ≤ 1

AVL Tree

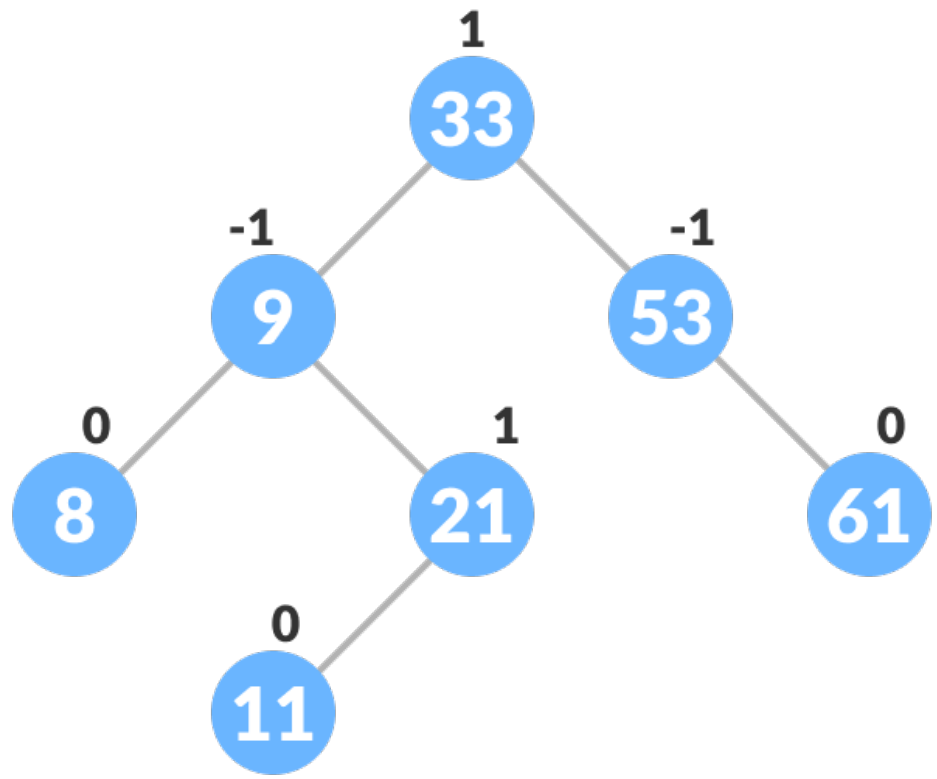
- **BF** =height(left subtree) – height(right subtree)
- Here we see that the first tree is balanced and the next two trees are not balanced −



- In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2.
- In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again.
- AVL tree permits difference (balance factor) to be only 1.

AVL Tree

- An example of a balanced AVL tree is:



AVL Tree

- Our goal is to balance BST.
    - Why.
        - Search time complexity:
            - Worst case: O(n)
            - In balance BST: O(log n)

Pros and Cons of AVL Trees

- Pros and Cons of AVL Trees
  1. Search, insertion and deletions are O(log n).
  2. The height balancing adds no more than a constant factor to the speed of insertion.
- Arguments against using AVL trees:
  1. Difficult to program and debug; need more space for balance factor.
  2. Asymptotically faster but rebalancing costs time.
  3. Most large searches are done in database systems on disk and useother structures (e.g. B-trees).
  4. May be OK to have O(N) for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).

- AVL Tree
- **Rotation**
- Insert
- Delete
- Max and Min
- Driver code
- Complexities of Different Operations

Create node

```
// AVL tree implementation in Java
// Create node
class Node {
    int item, height;
    Node left, right;

    Node(int d) {
        item = d;
        height = 1;
    }
}
```

Create node

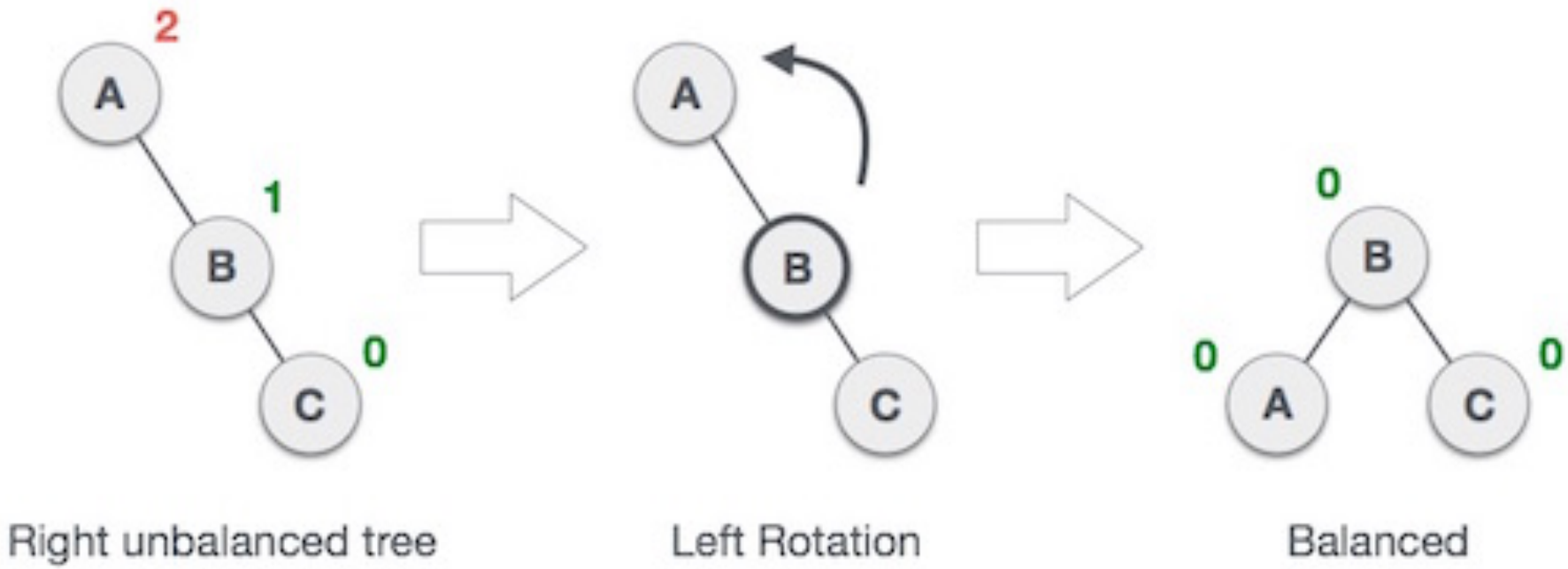
```
int height(Node N) {
    if (N == null) return 0;
    return N.height;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}

// Get balance factor of a node
int getBalanceFactor(Node N) {
    if (N == null) return 0;
    return height(N.left) - height(N.right);
}
```

AVL Rotation

- If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.
- To balance itself, an AVL tree may perform the following four kinds of rotations −
  - Left rotation
  - Right rotation
  - Left-Right rotation
  - Right-Left rotation

Left Rotation

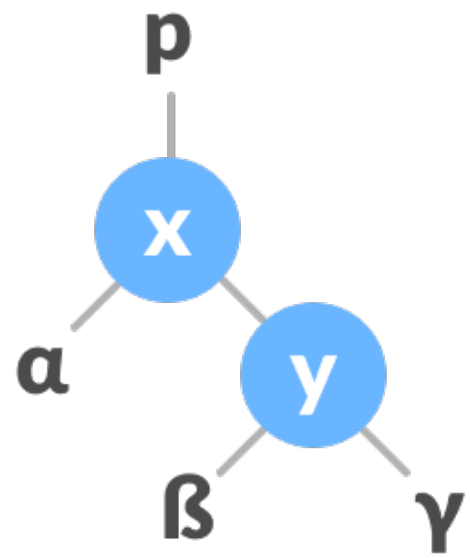
- If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



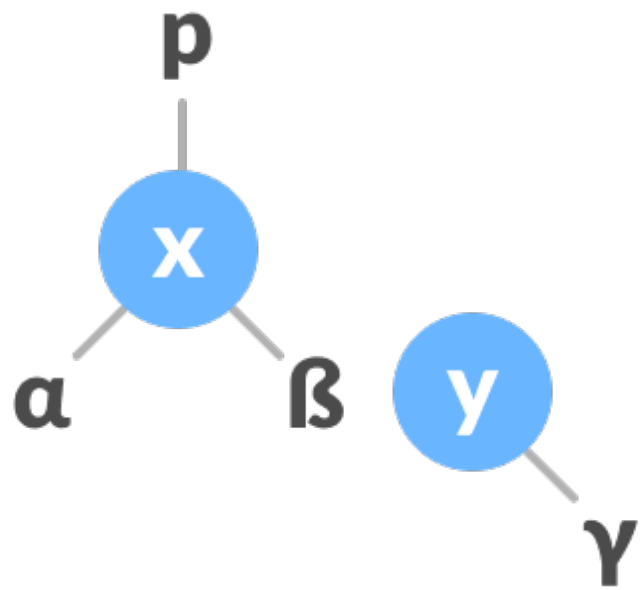
- In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree.
- We perform the left rotation by making **A** the left-subtree of B.

Algorithm of Left Rotation

- Let the initial tree be:



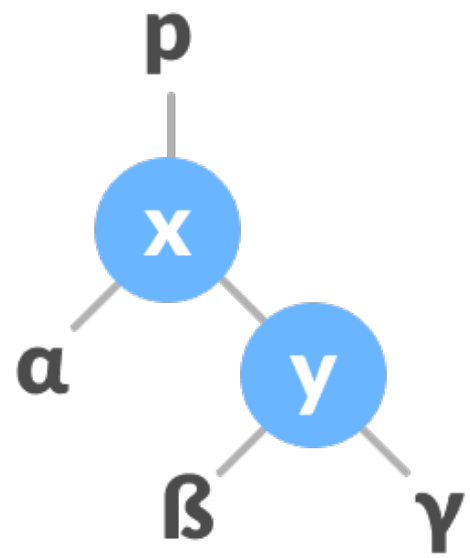
- If y has a left subtree, assign x as the parent of the left subtree of y.



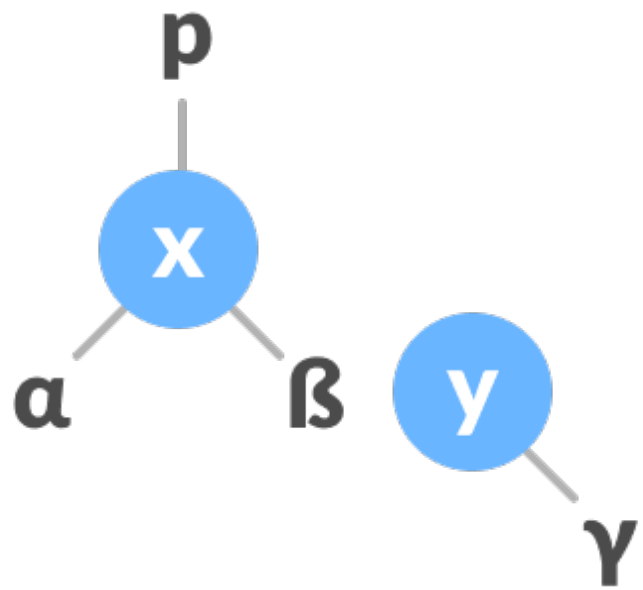
Assign x as the parent of the left subtree of y

Algorithm of Left Rotation

- Let the initial tree be:



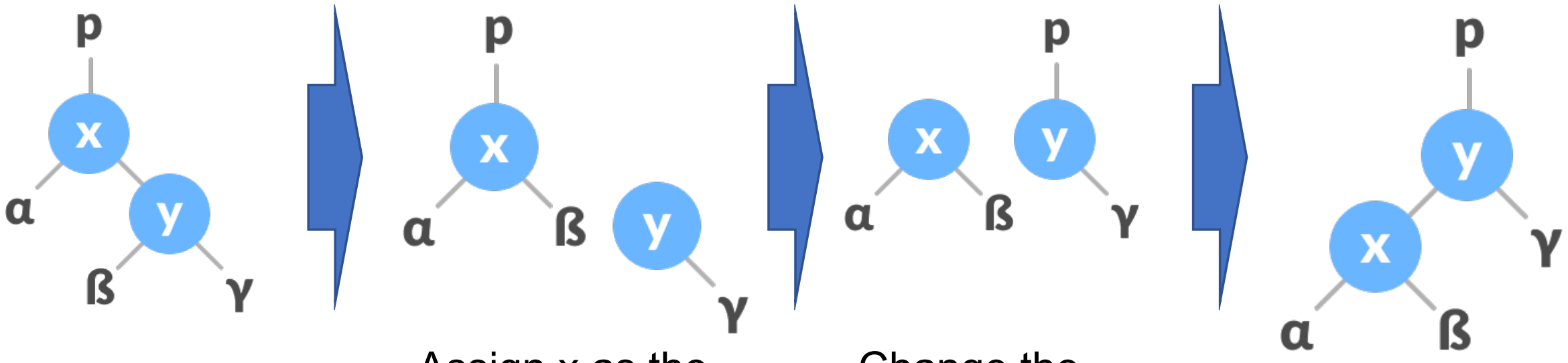
- **Step 1.** If y has a left subtree, assign x as the parent of the left subtree of y.



Assign x as the parent of the left subtree of y

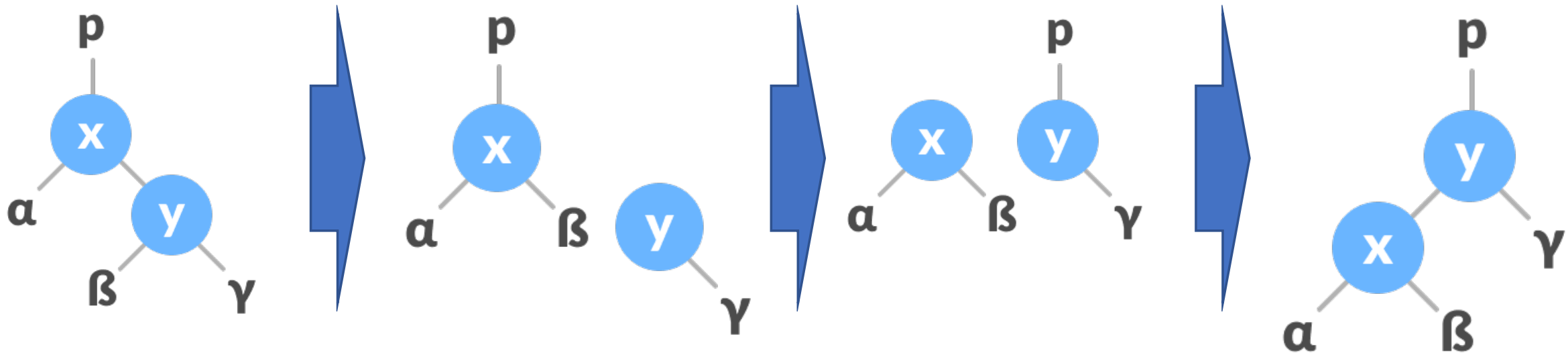
Algorithm of Left Rotation

- **Step 1.** If y has a left subtree, assign x as the parent of the left subtree of y.
- **Step 2.** If the parent of x is NULL, make y as the root of the tree.
- **Step 3.** Else if x is the left child of p, make y as the left child of p.
- **Step 4.** Else assign y as the right child of p.
- **Step 5.** Make y as the parent of x.



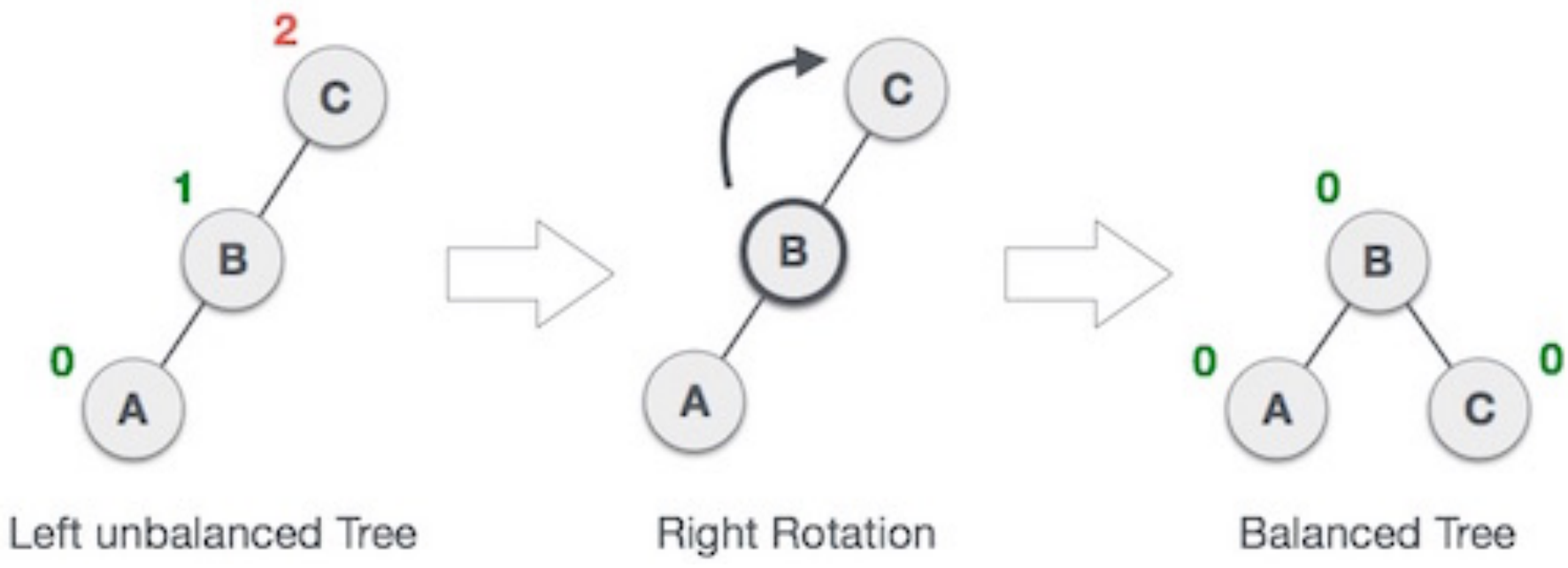
Left Rotation

```
Node leftRotate(Node x) {
    Node y = x.right;
    Node T2 = y.left;
    y.left = x;
    x.right = T2;
    x.height = max(height(x.left), height(x.right)) + 1;
    y.height = max(height(y.left), height(y.right)) + 1;
    return y;
}
```



Right Rotation

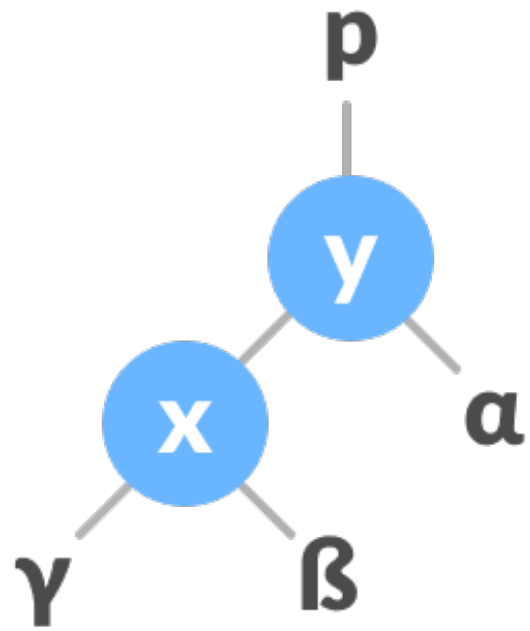
- AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree.
- The tree then needs a right rotation.



- As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

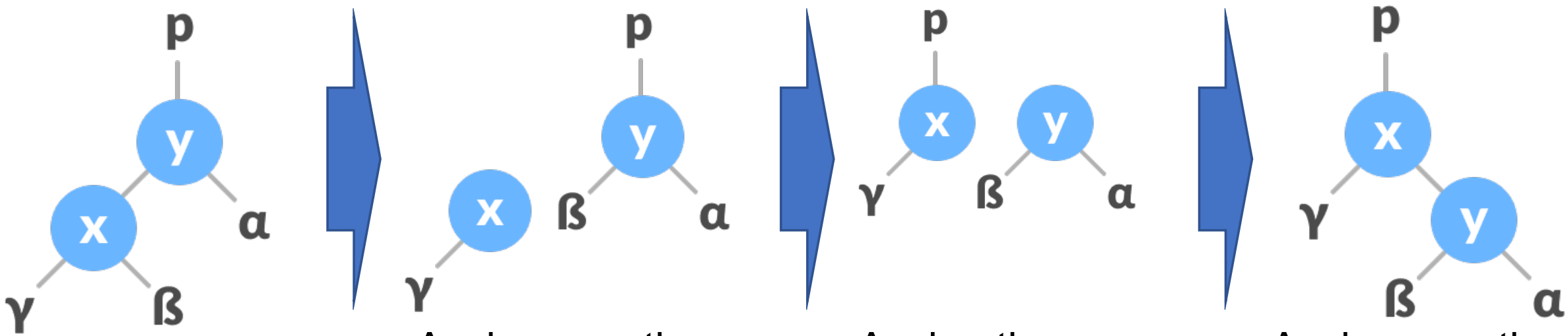
Algorithm of Right Rotation

- Let the initial tree be:



Algorithm of Right Rotation

- **Step 1.** If x has a right subtree, assign y as the parent of the right subtree of x.
- **Step 2.** If the parent of y is NULL, make x as the root of the tree.
- **Step 3.** Else if y is the right child of its parent p, make x as the right child of p.
- **Step 4.** Else assign x as the left child of p.
- **Step 5.** Make x as the parent of y.



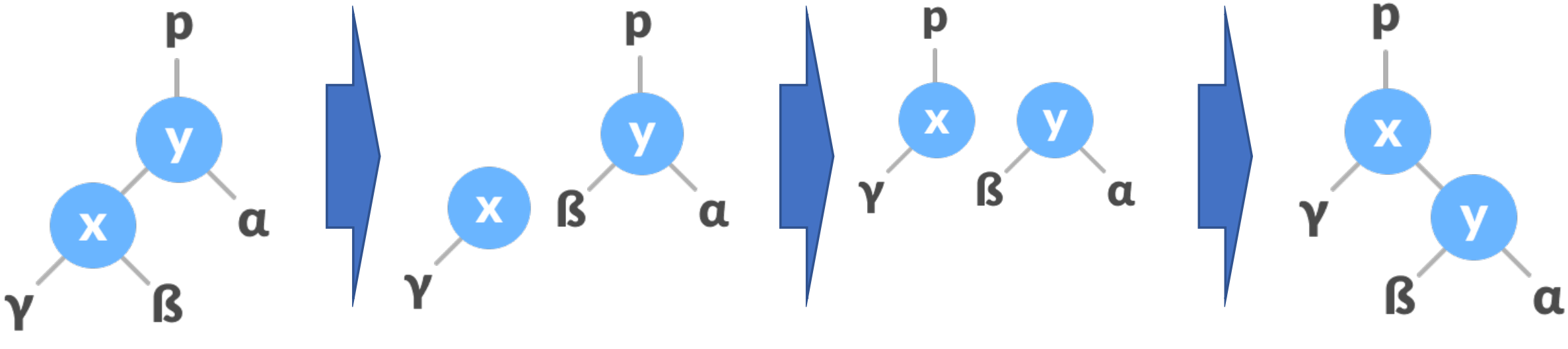
Assign y as the parent of the right subtree of x

Assign the parent of y as the parent of x.

Assign x as the parent of y

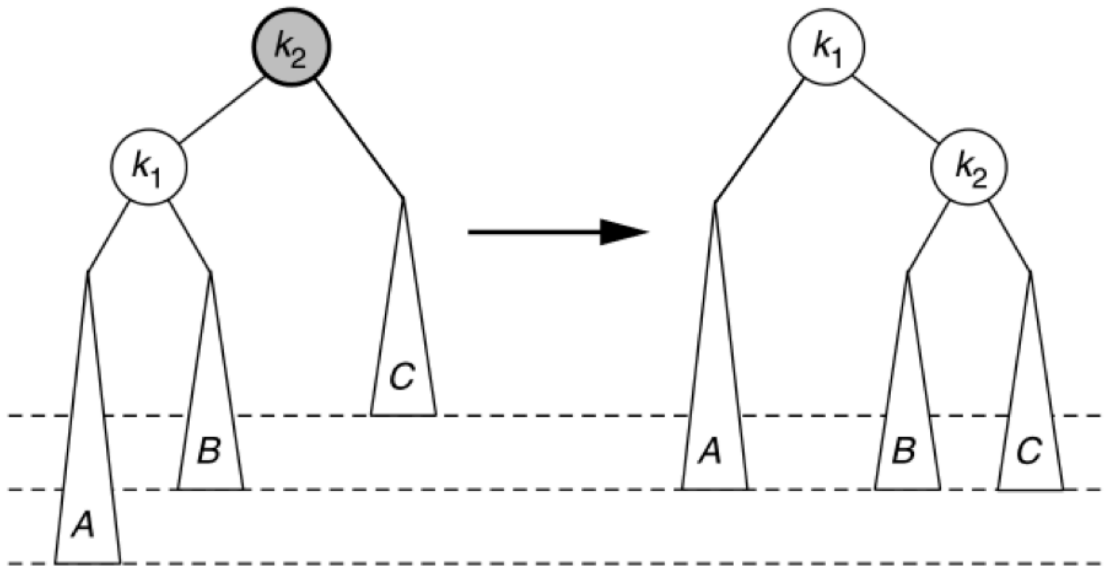
Right Rotation

```
Node rightRotate(Node y) {
    Node x = y.left;
    Node T2 = x.right;
    x.right = y;
    y.left = T2;
    y.height = max(height(y.left), height(y.right)) + 1;
    x.height = max(height(x.left), height(x.right)) + 1;
    return x;
}
```



Right Rotation

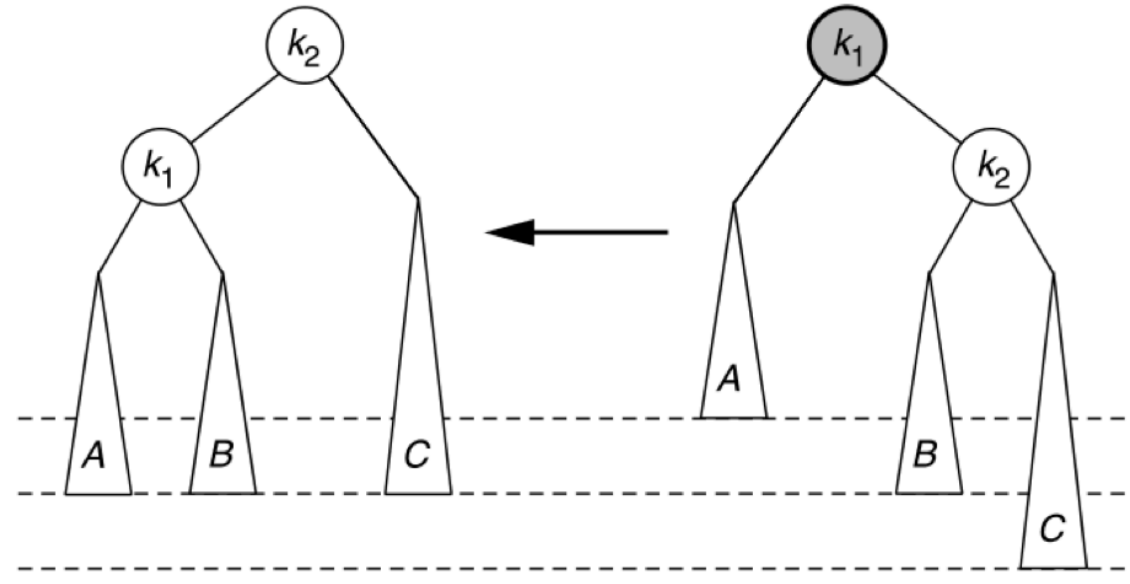
Rotation node becomes the right subtree



(a) Before rotation (b) After rotation

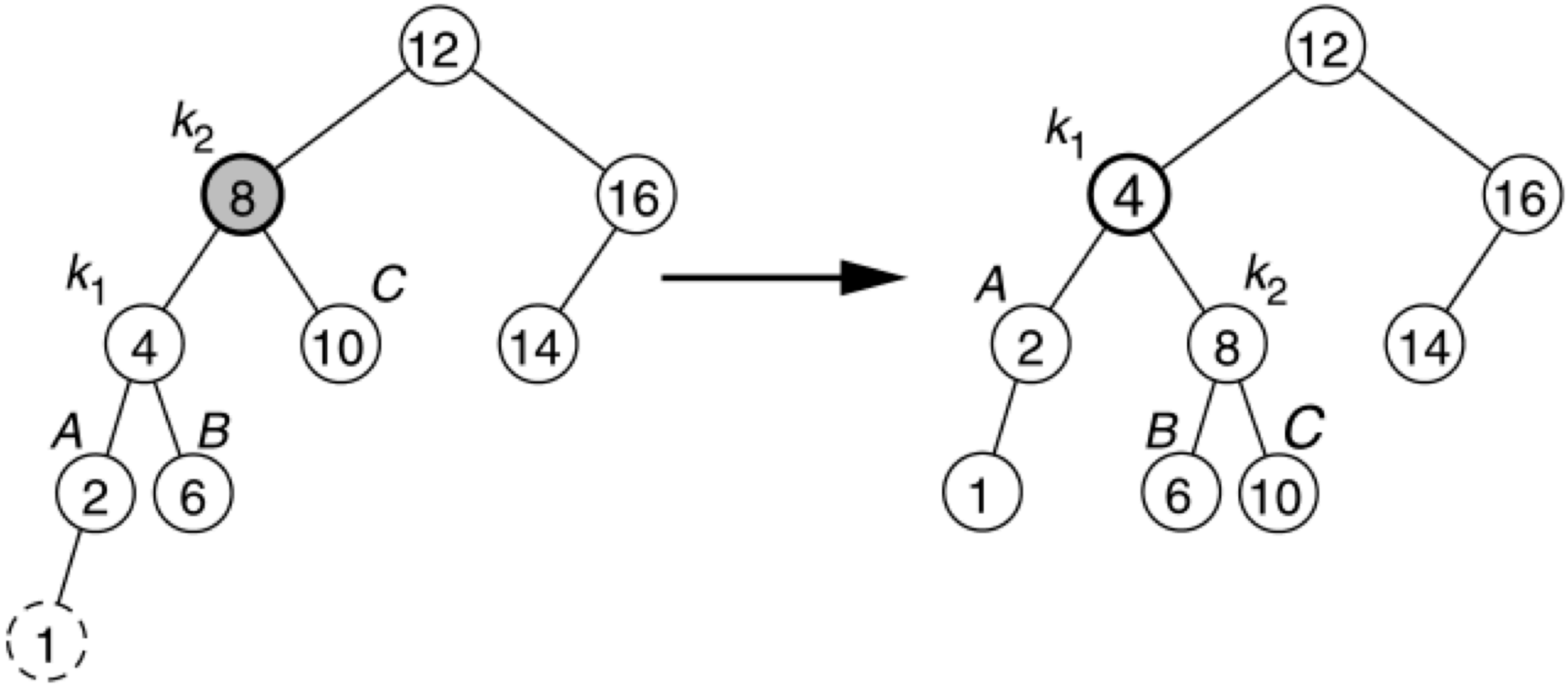
Left Rotation

Rotation node becomes the left subtree



(a) After rotation (b) Before rotation

Fixing an Insertion with a Single Rotation

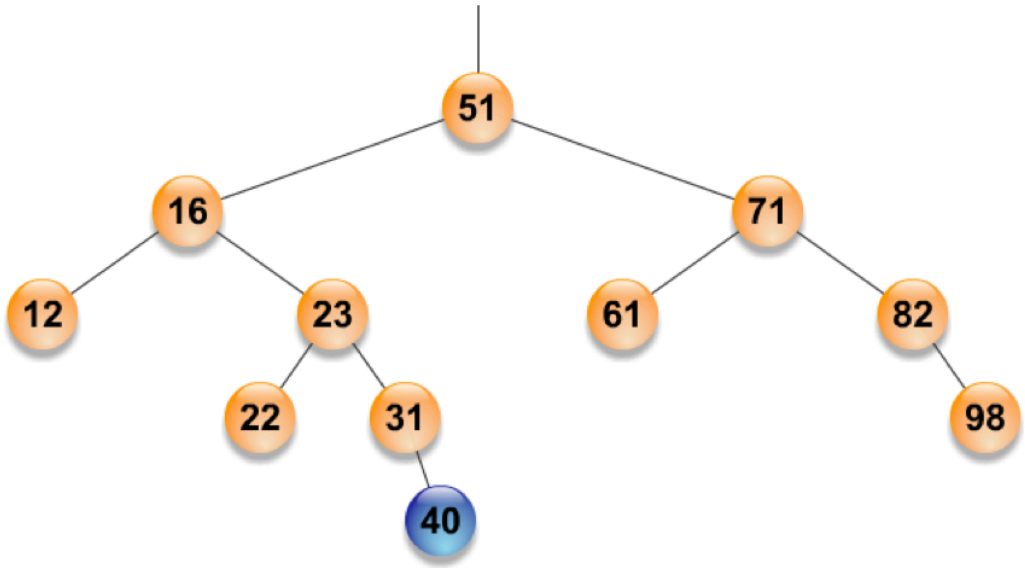


(a) Before rotation

(b) After rotation

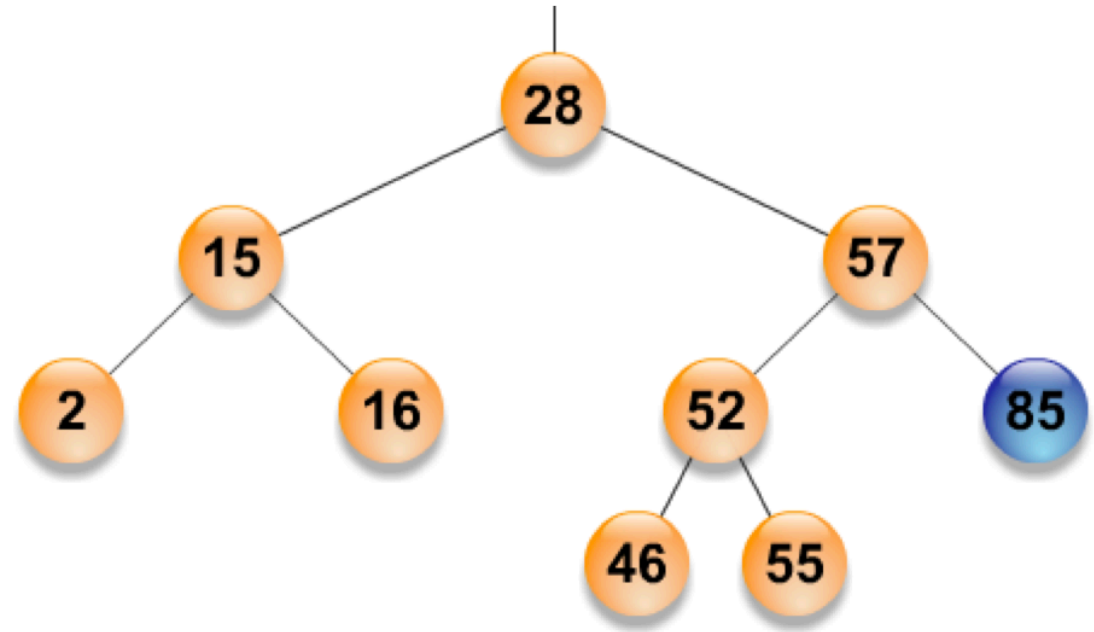
Problem 1

- 40 was just inserted
- Rebalance tree rooted at 16
- Left-rotate 16



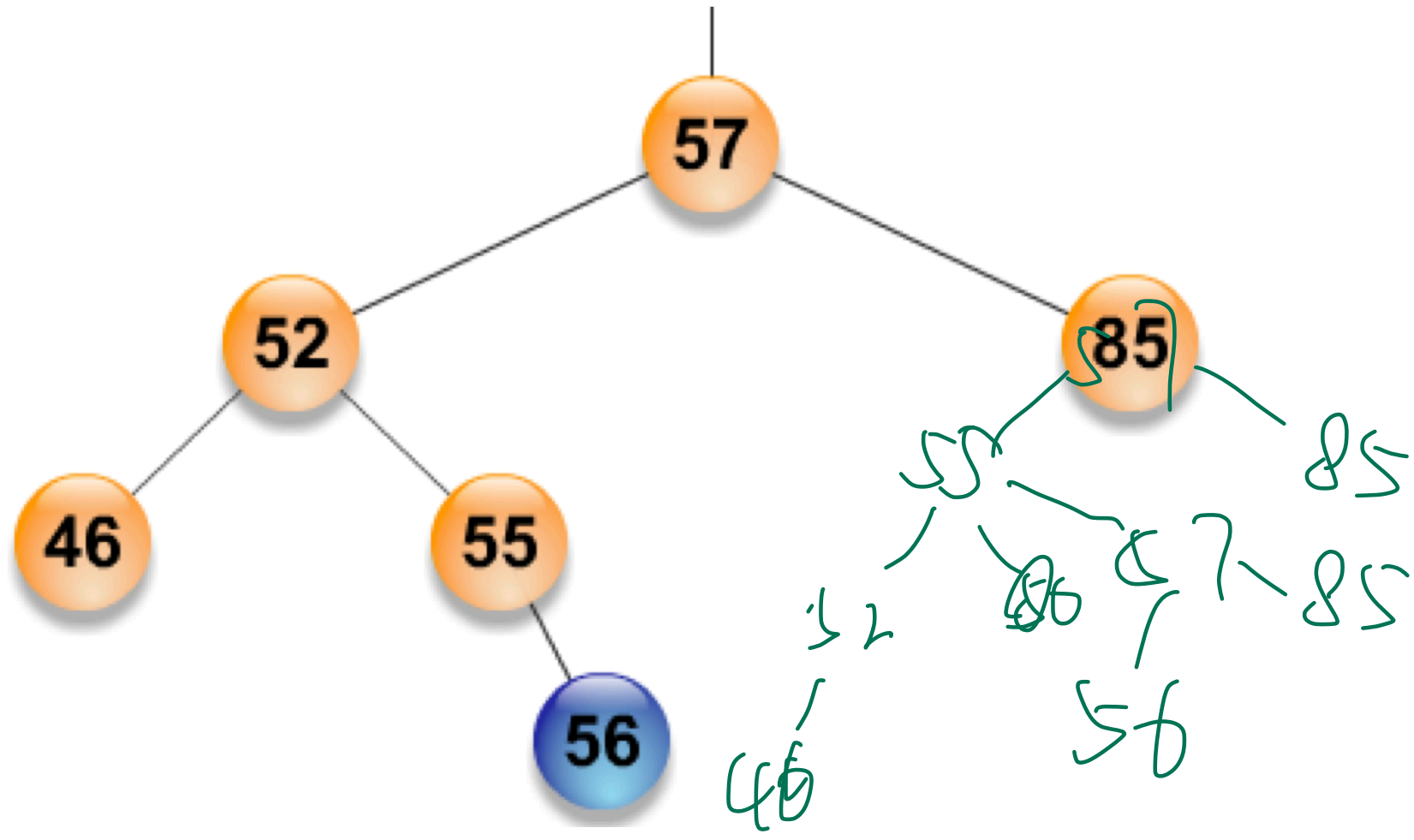
Problem 2

- 85 is being removed
- Rebalance tree rooted at 57
- Right rotate 57

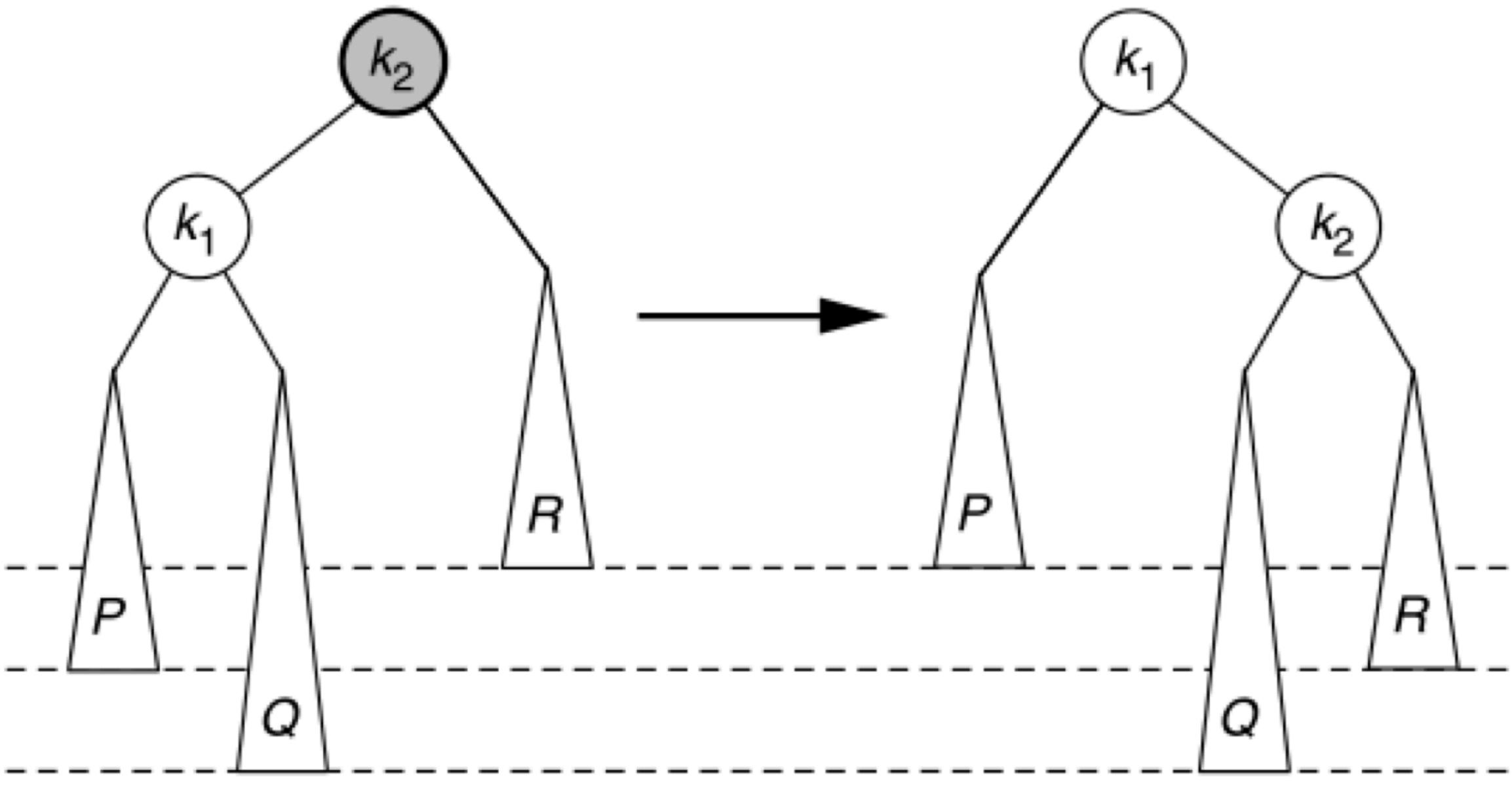


Question: Can this be fixed with single rotation?

56 was just inserted : restore AVL property with a single rotation?



Cannot fix following class of situations with a single rotation

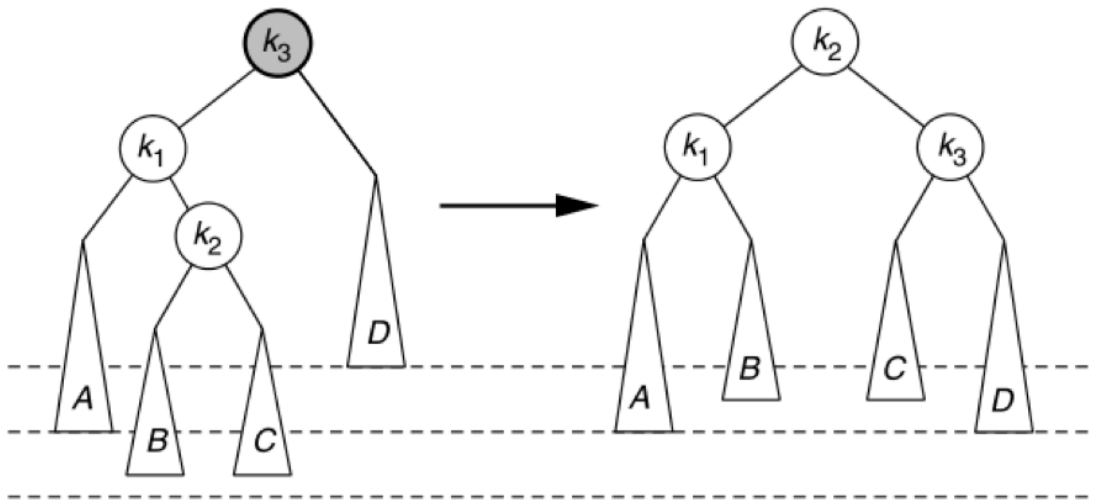


(a) Before rotation

(b) After rotation

Left-Right

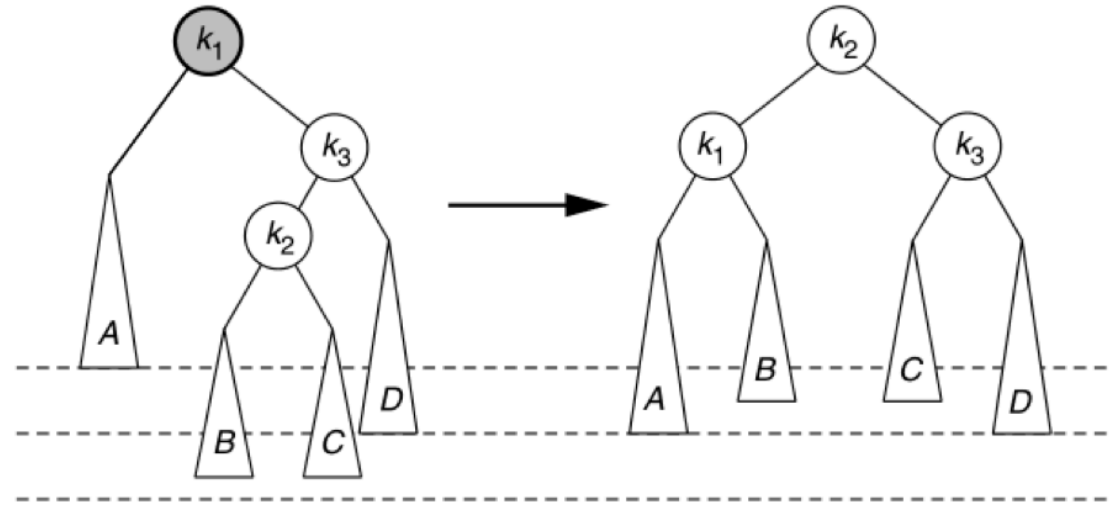
- Left Rotate at k_1
- Right-rotate at k_3



(a) Before rotation (b) After rotation

Right-Left

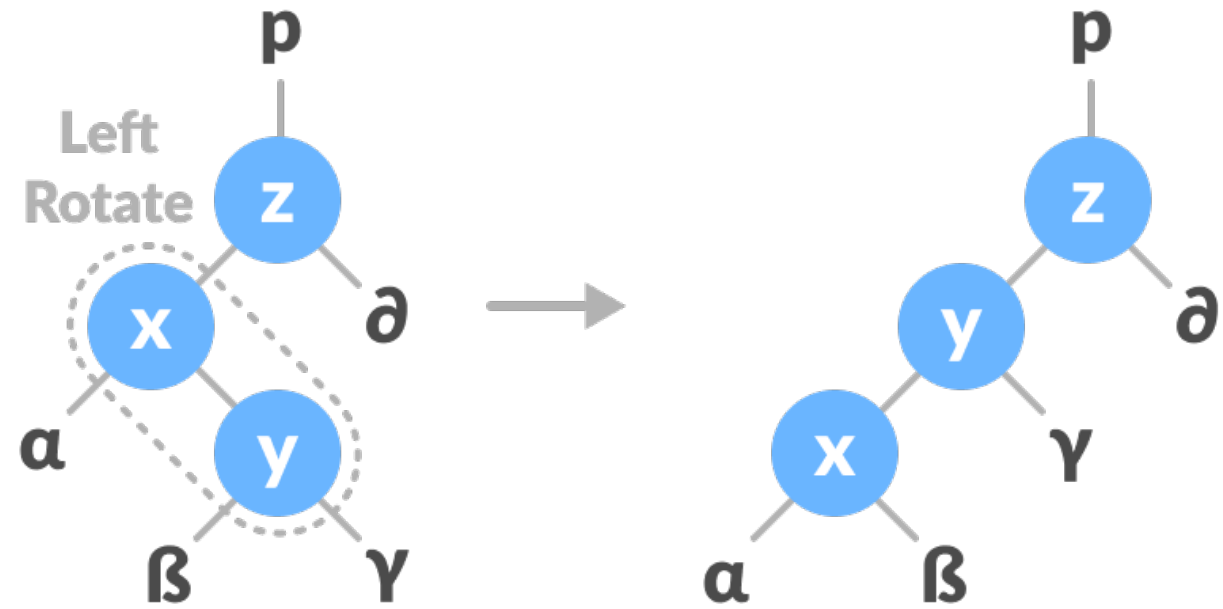
- Right Rotate at k_3
- Left Rotate at k_2



(a) Before rotation (b) After rotation

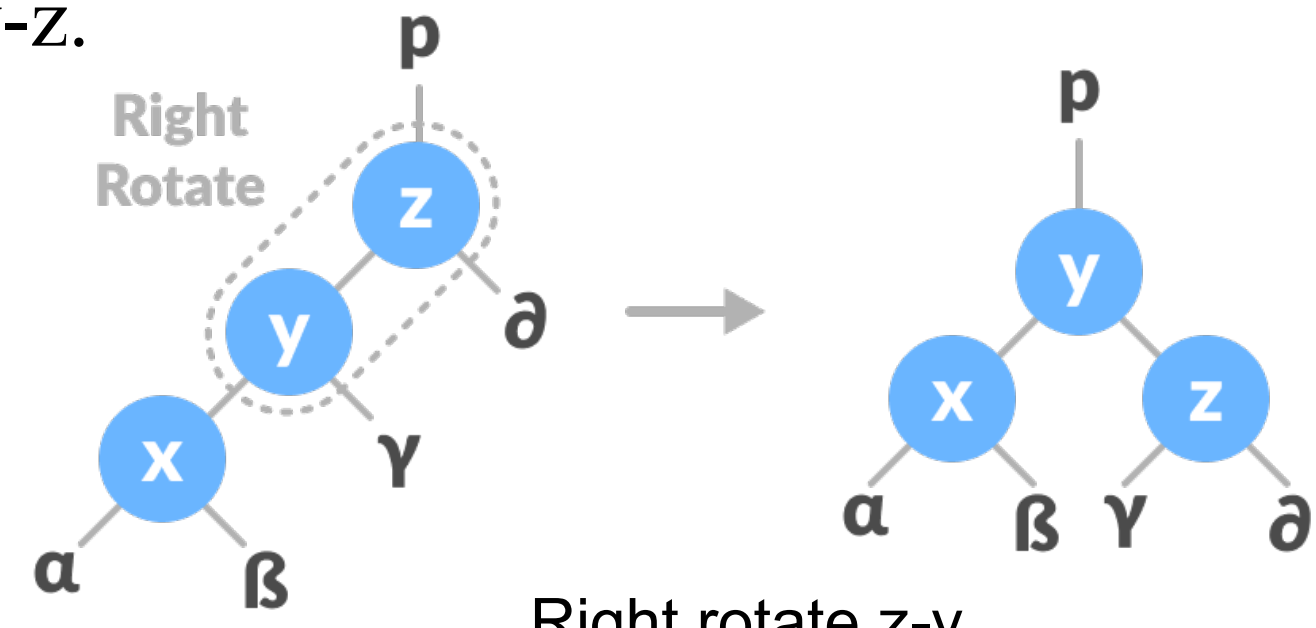
Left-Right and Right-Left Rotation

- In left-right rotation, the arrangements are first shifted to the left and then to the right.
- **Step 1.** Do left rotation on x-y.



Left rotate x-y

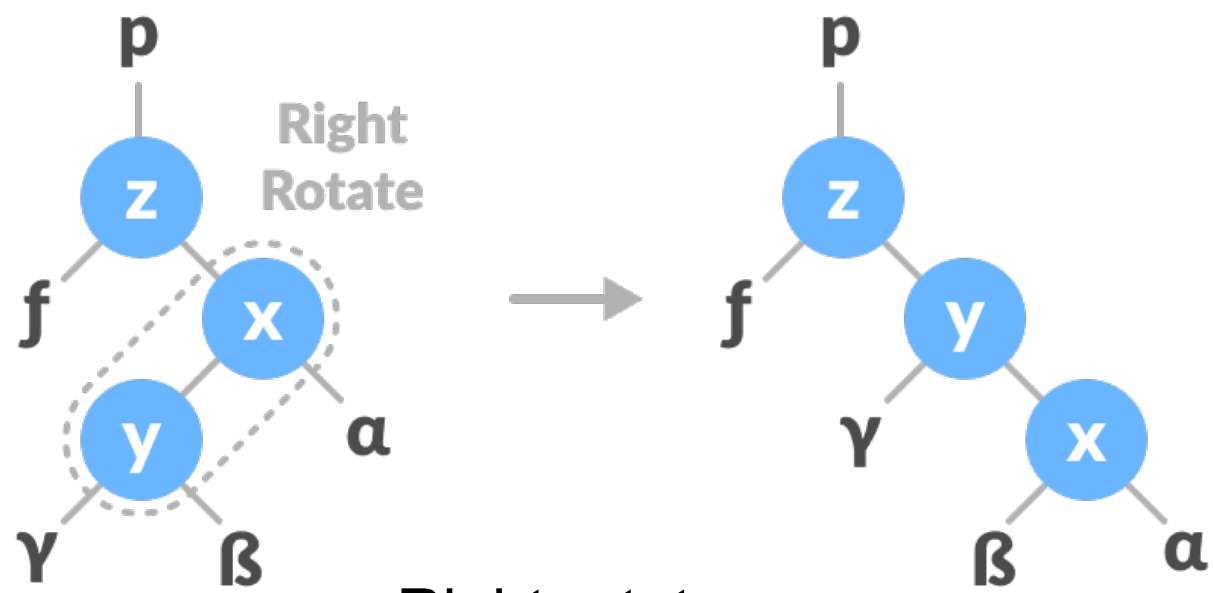
- **Step 2.** Do right rotation on y-z.



Right rotate z-y

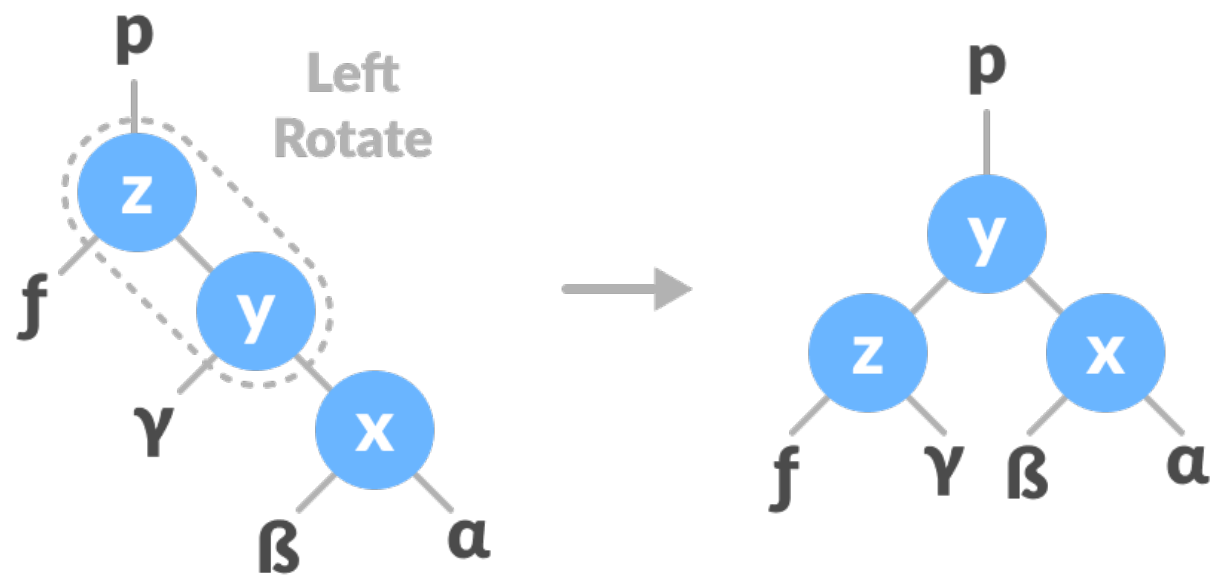
Left-Right and Right-Left Rotation

- In right-left rotation, the arrangements are first shifted to the right and then to the left.
- **Step 1.** Do right rotation on x-y.



Right rotate x-y

- **Step 2.** Do left rotation on y-z.

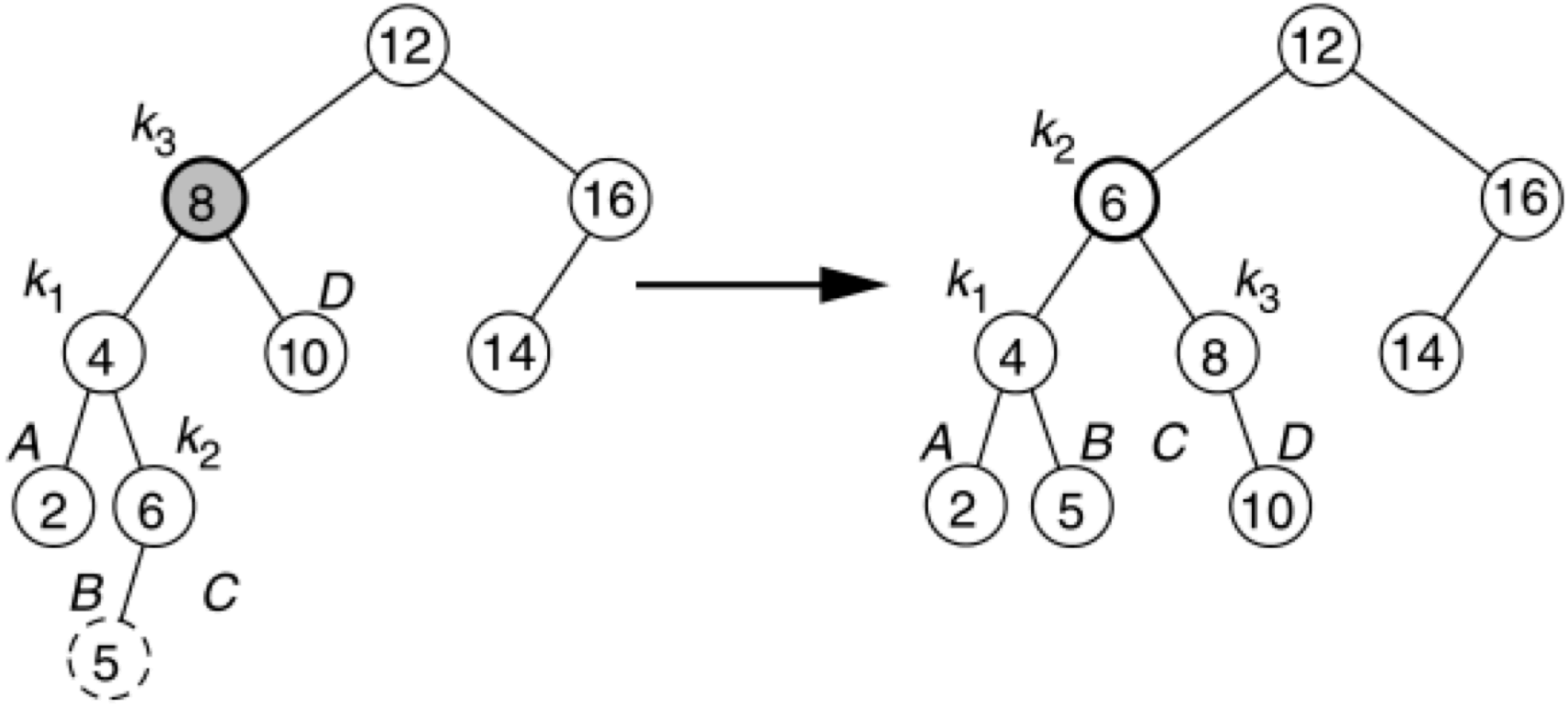


Left rotate z-y

Fixing an Insertion with a Double Rotation

Insert 5, perform two rotations to balance heights

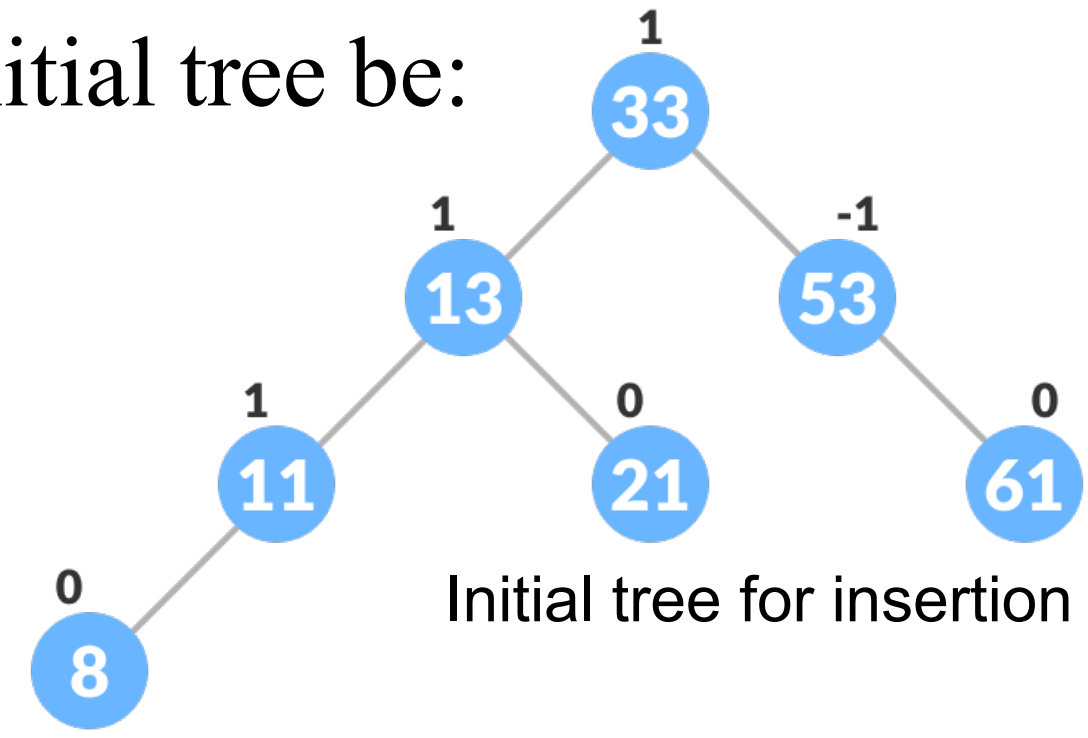
- Problem is at 8: left height 3, right height 1
- Left rotate 4 (height imbalance remains)
- Right rotate 8 (height imbalance fixed)



- AVL Tree
- Rotation
- **Insert**
- Delete
- Max and Min
- Driver code
- Complexities of Different Operations

Algorithm to insert a newNode

- A newNode is always inserted as a **leaf** node with balance factor equal to 0.
- Let the initial tree be:



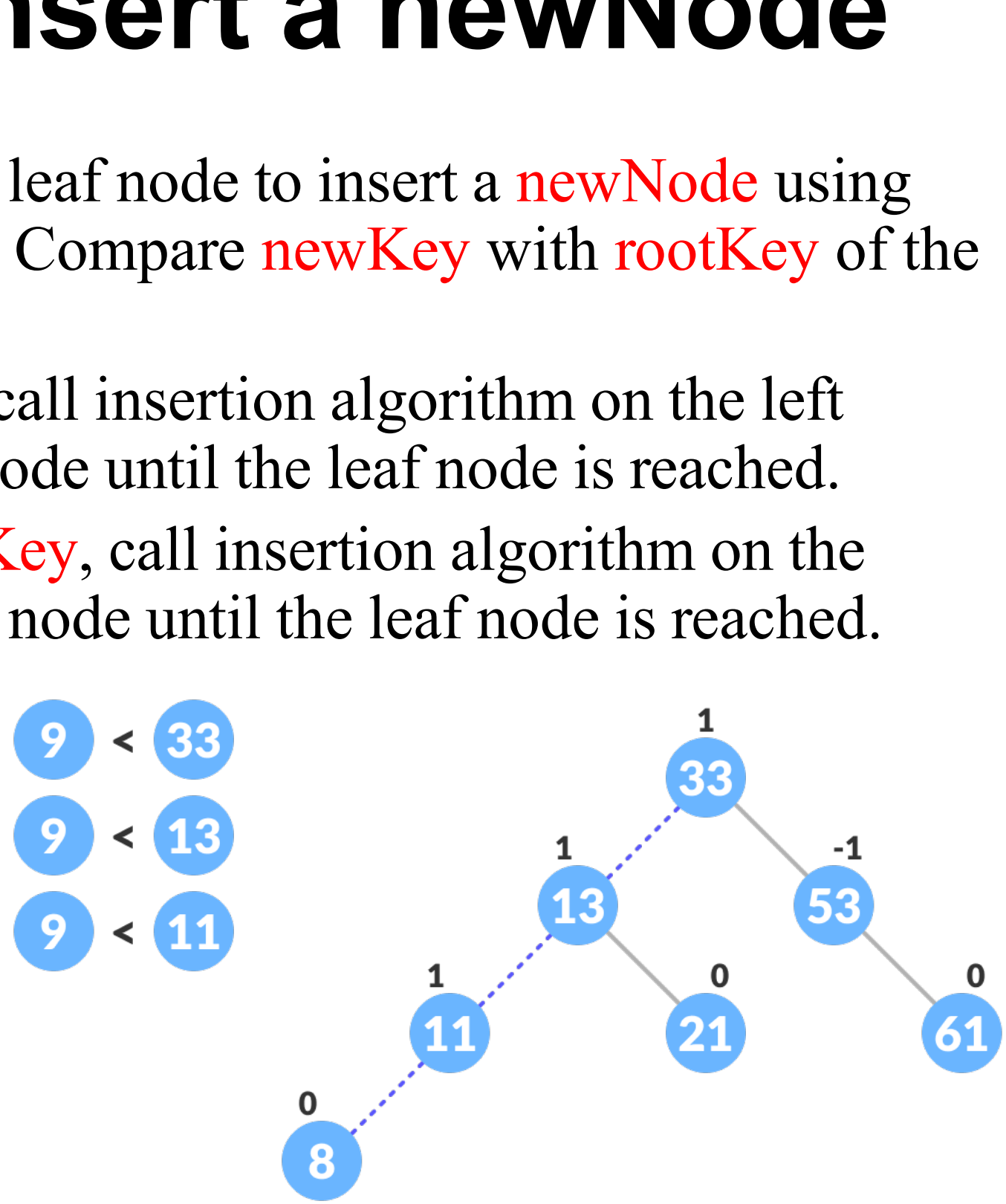
Initial tree for insertion

- Let the node, 9, to be inserted be:

New node

Algorithm to insert a newNode

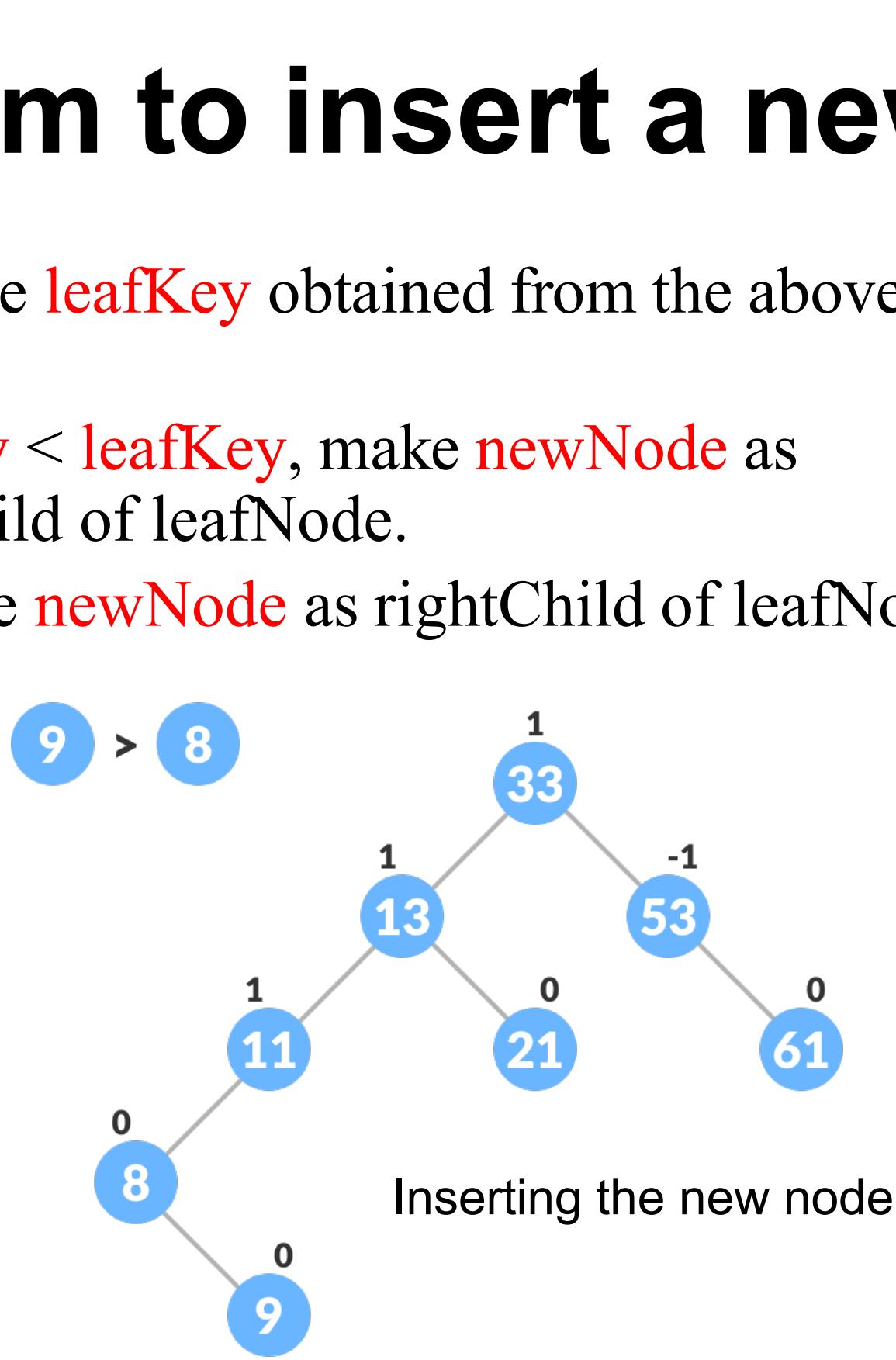
- **Step 1.** Go to the appropriate leaf node to insert a newNode using the following recursive steps. Compare newKey with rootKey of the current tree.
  a) If newKey < rootKey, call insertion algorithm on the left subtree of the current node until the leaf node is reached.
  b) Else if newKey > rootKey, call insertion algorithm on the right subtree of current node until the leaf node is reached.
  c) Else, return leafNode.



Finding the location to insert newNode

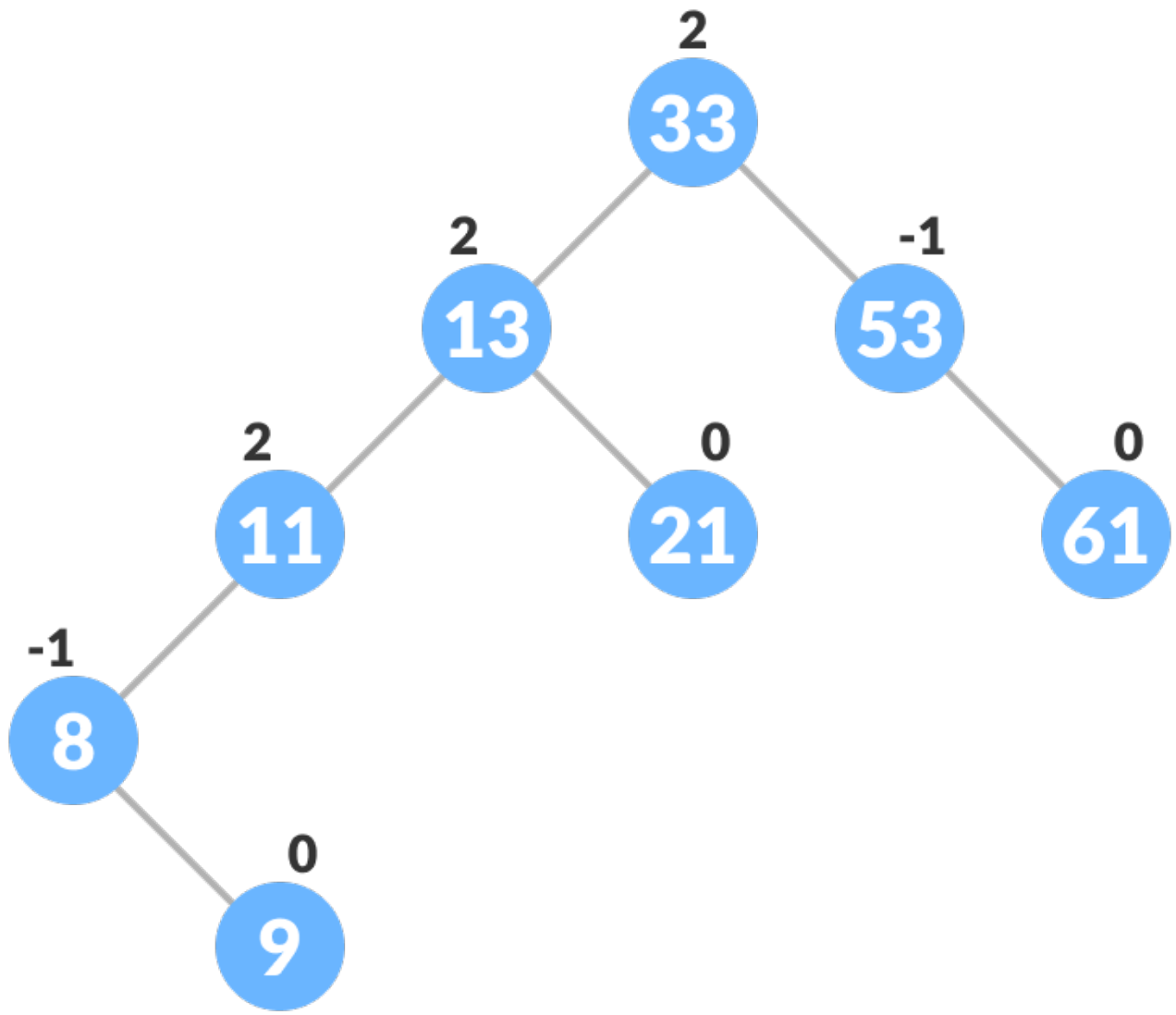
Algorithm to insert a newNode

- **Step 2.** Compare leafKey obtained from the above steps with newKey:
  a) If newKey < leafKey, make newNode as the leftChild of leafNode.
  b) Else, make newNode as rightChild of leafNode.



Algorithm to insert a newNode

- **Step 3.** Update balanceFactor of the nodes.



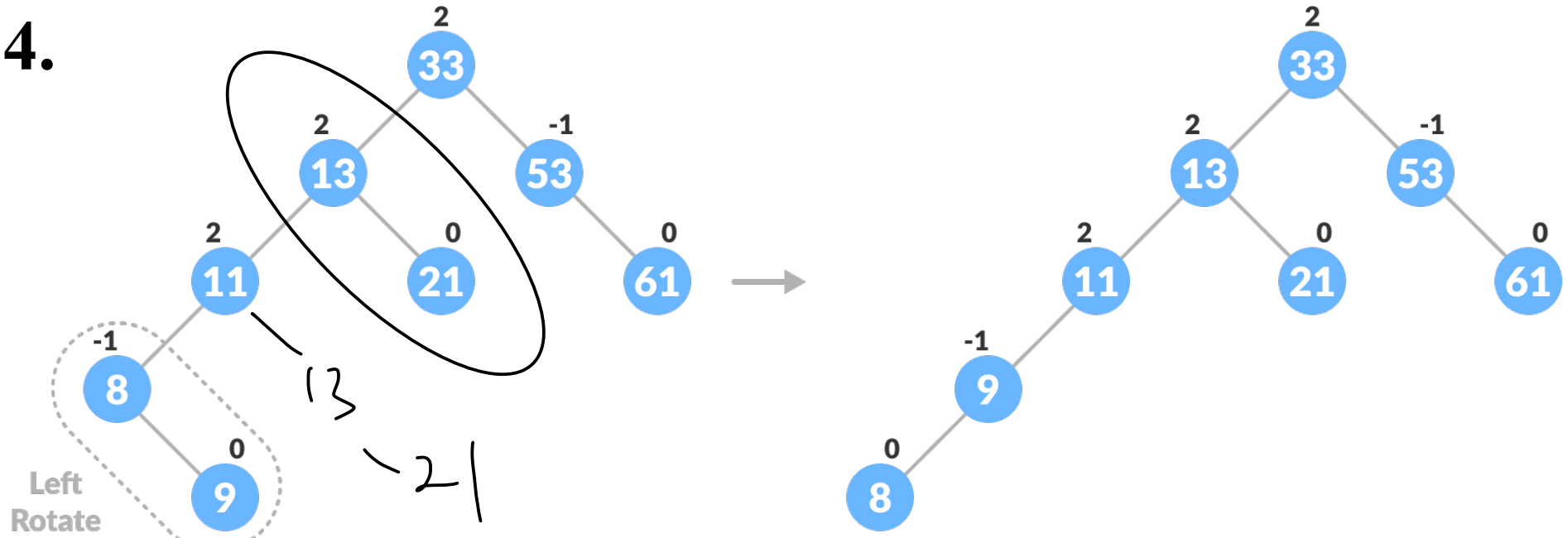
Updating the balance factor after insertion

Algorithm to insert a newNode

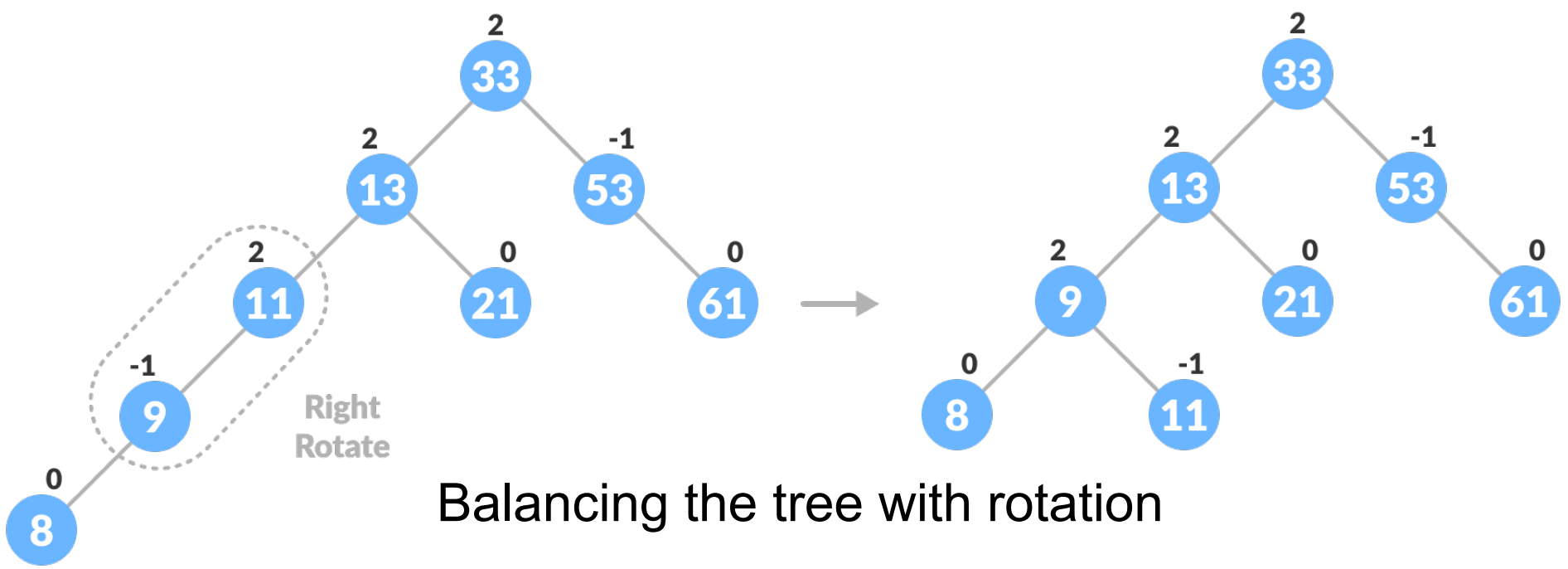
- **Step 4.** If the nodes are unbalanced, then rebalance the node.
  a) If balanceFactor > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation
     i. If newNodeKey < leftChildKey do right rotation.
     ii. Else, do left-right rotation.
  b) If balanceFactor < -1, it means the height of the right subtree is greater than that of the left subtree. So, do right rotation or right-left rotation
     i. If newNodeKey > rightChildKey do left rotation.
     ii. Else, do right-left rotation

Algorithm to insert a newNode

- **Step 4.**



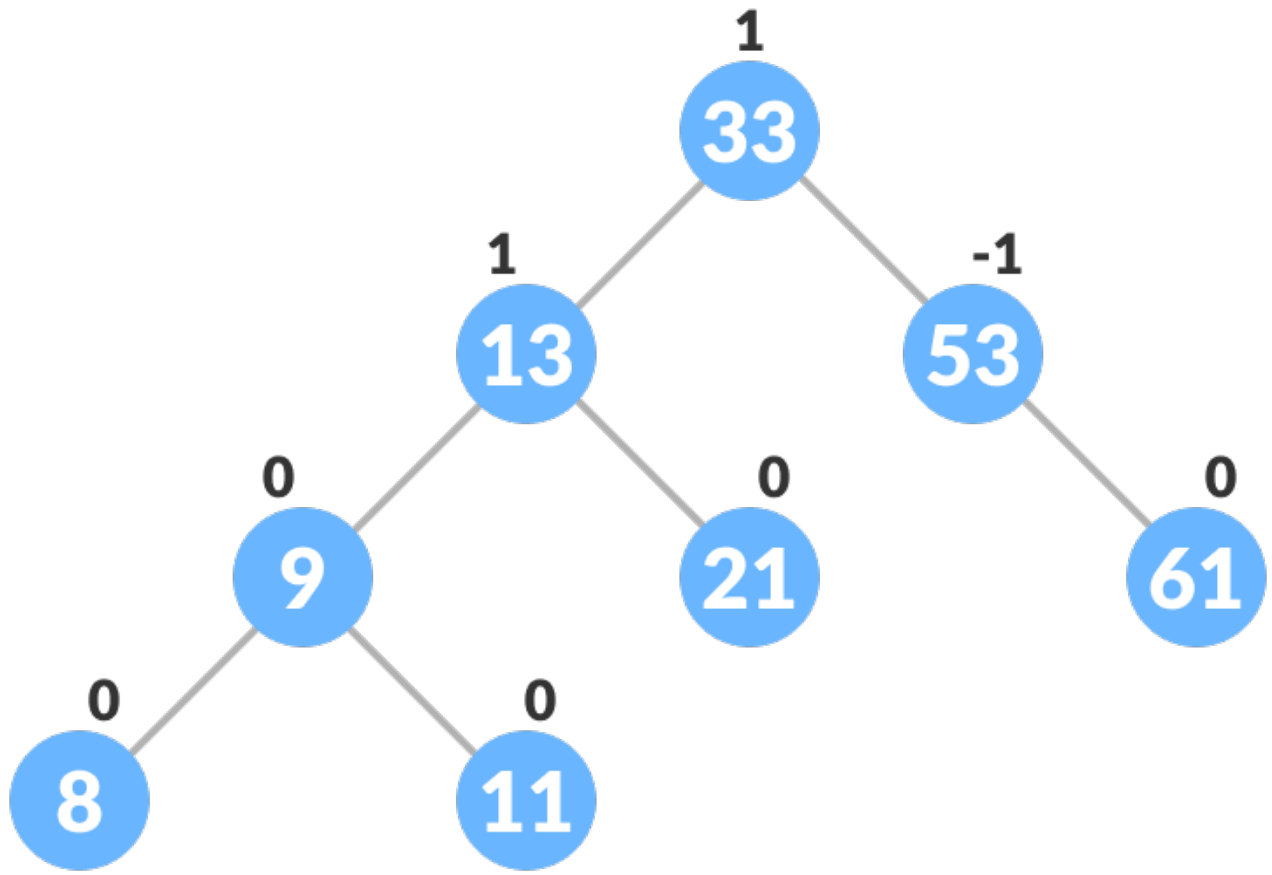
Balancing the tree with rotation



Balancing the tree with rotation

Algorithm to insert a newNode

- The final tree is:



Final balanced tree

Insert a node

```
Node insertNode(Node node, int item) {
// Find the position and insert the node
    if (node == null) return (new Node(item));
    if (item < node.item) node.left = insertNode(node.left, item);
    else if (item > node.item) node.right = insertNode(node.right, item);
    else return node;
// Update the balance factor of each node. And, balance the tree
    node.height = 1 + max(height(node.left), height(node.right));
    int balanceFactor = getBalanceFactor(node);
    if (balanceFactor > 1) {
        if (item < node.left.item) { return rightRotate(node); }
        else if (item > node.left.item) {
            node.left = leftRotate(node.left);
            return rightRotate(node); } }
        if (balanceFactor < -1) {
            if (item > node.right.item) { return leftRotate(node); }
            else if (item < node.right.item) {
                node.right = rightRotate(node.right);
                return leftRotate(node); }
    }
    return node;
}
```

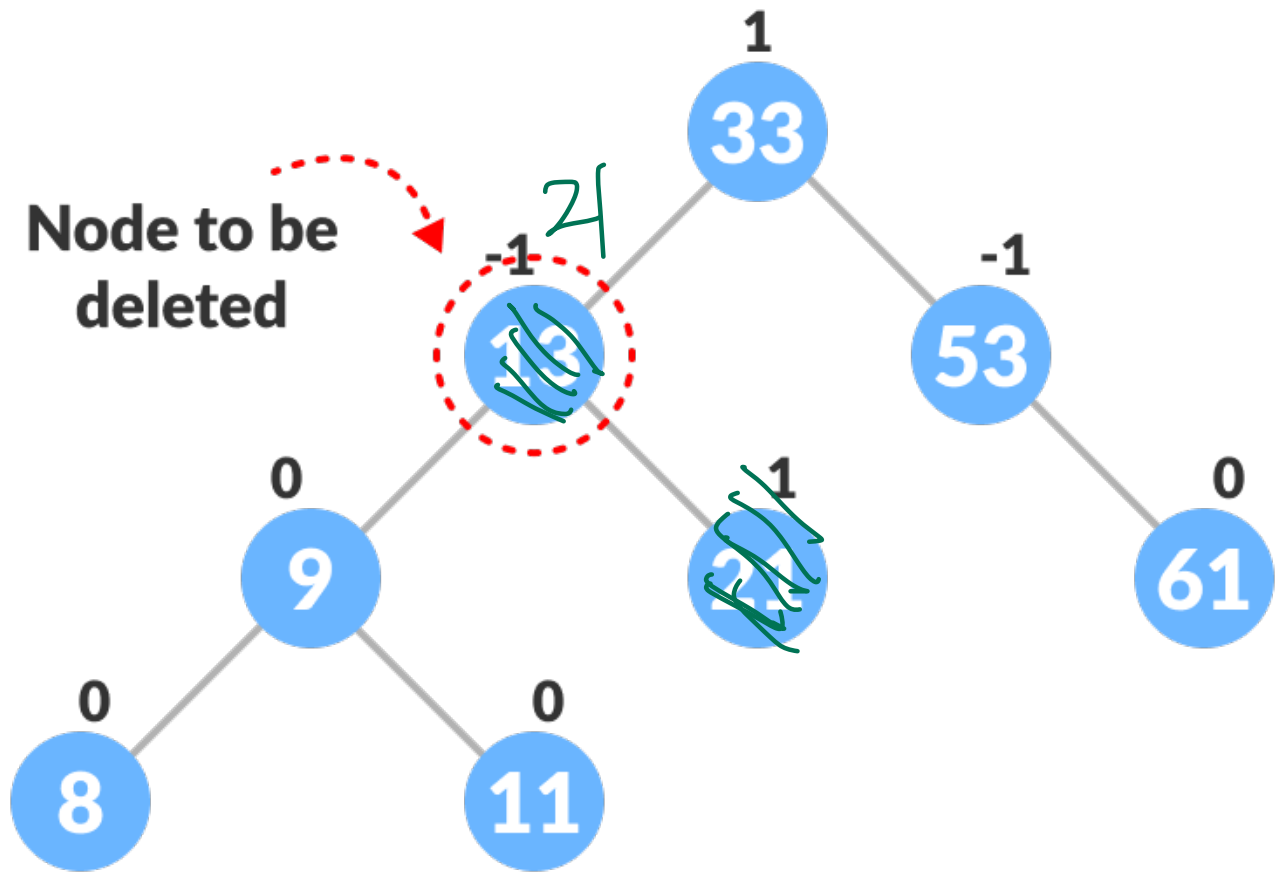
- AVL Tree
- Rotation
- Insert
- **Delete**
- Max and Min
- Driver code
- Complexities of Different Operations

Algorithm to Delete a node

- A node is always deleted as a leaf node.
- After deleting a node, the balance factors of the nodes get changed.
- In order to rebalance the balance factor, suitable rotations are performed.

Algorithm to Delete a node

- Locate nodeToBeDeleted (recursion is used to find nodeToBeDeletedin the code used below).



Locating the node to be deleted

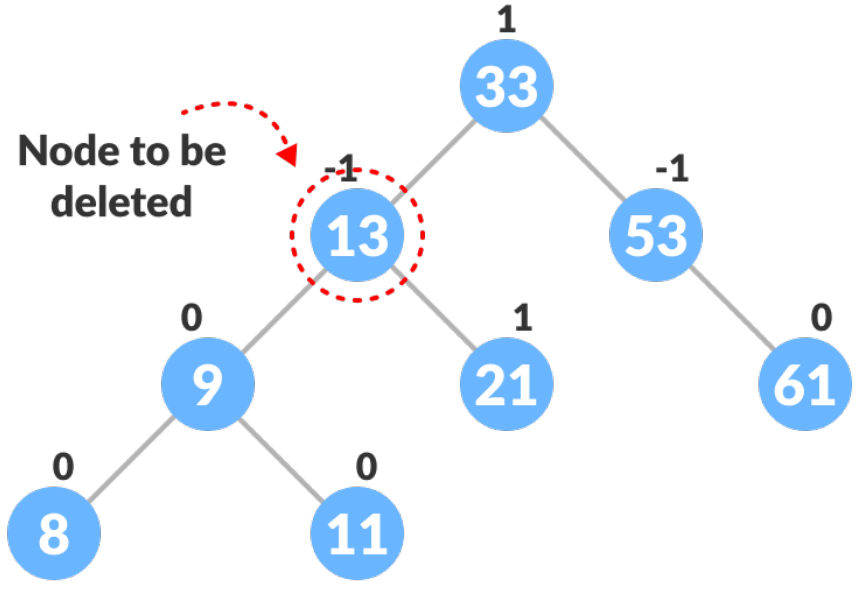
Algorithm to Delete a node

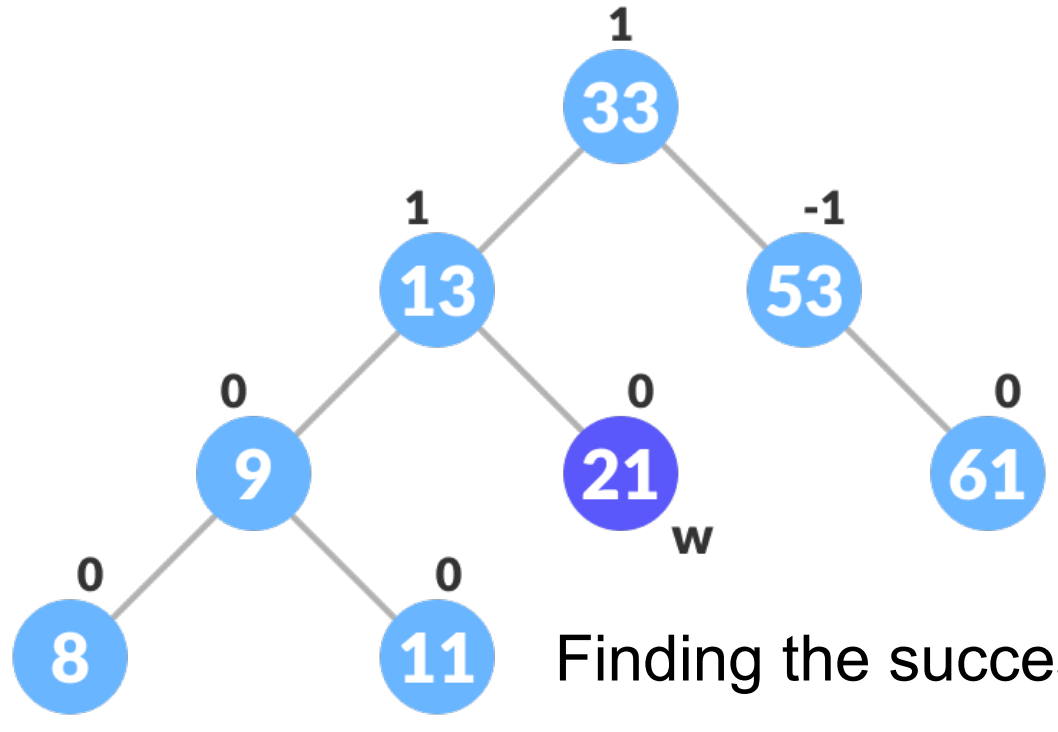
- There are three cases for deleting a node:

  Step 1. If nodeToBeDeleted is the **leaf node** (ie. does not have any child), then remove nodeToBeDeleted.

  Step 2. If nodeToBeDeleted **has one child**, then substitute the contents of nodeToBeDeleted with that of the child. Remove the child.

  Step 3. If nodeToBeDeleted **has two children**, find the inorder successor w of nodeToBeDeleted (ie. node with a minimum value of key in the right subtree).





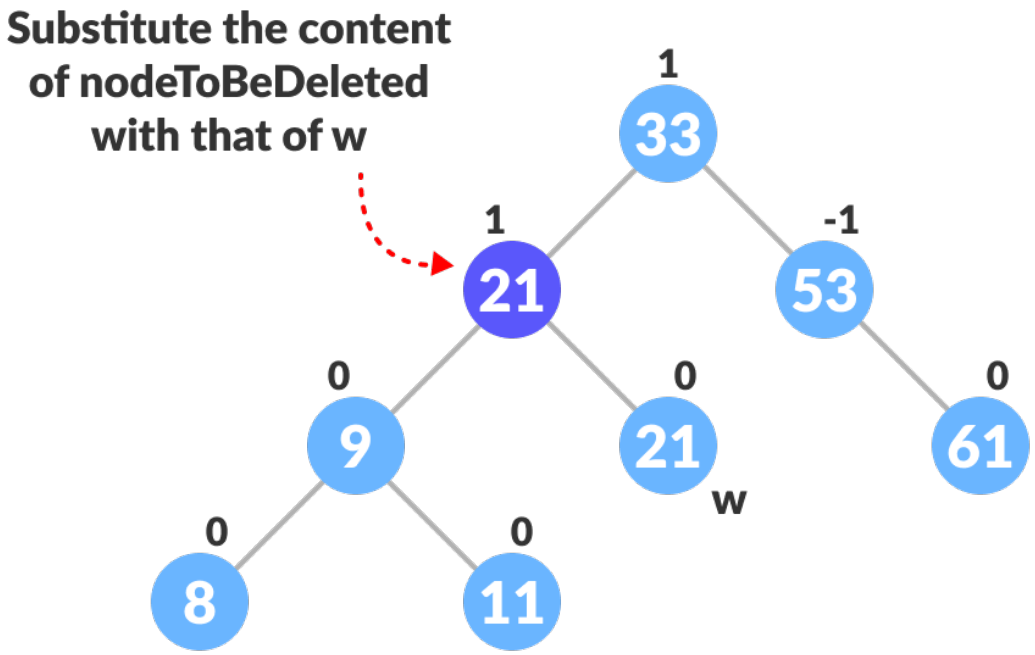
Finding the successor

Algorithm to Delete a node

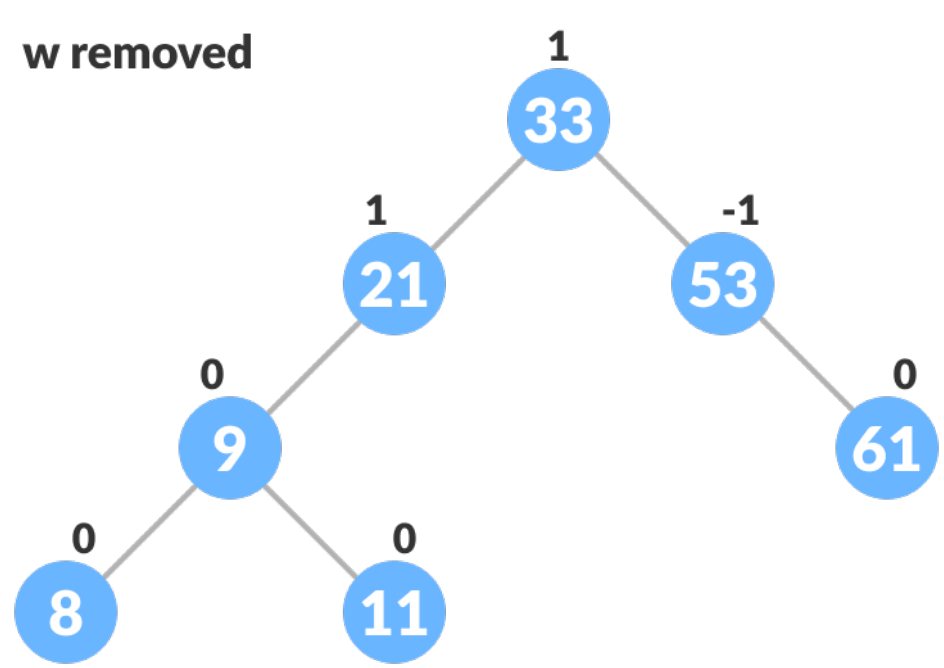
Step 3. If nodeToBeDeleted **has two children**, find the inorder successor w of nodeToBeDeleted (ie. node with a minimum value of key in the right subtree).

i. Substitute the contents of nodeToBeDeleted with that of w.

ii. Remove the leaf node w.



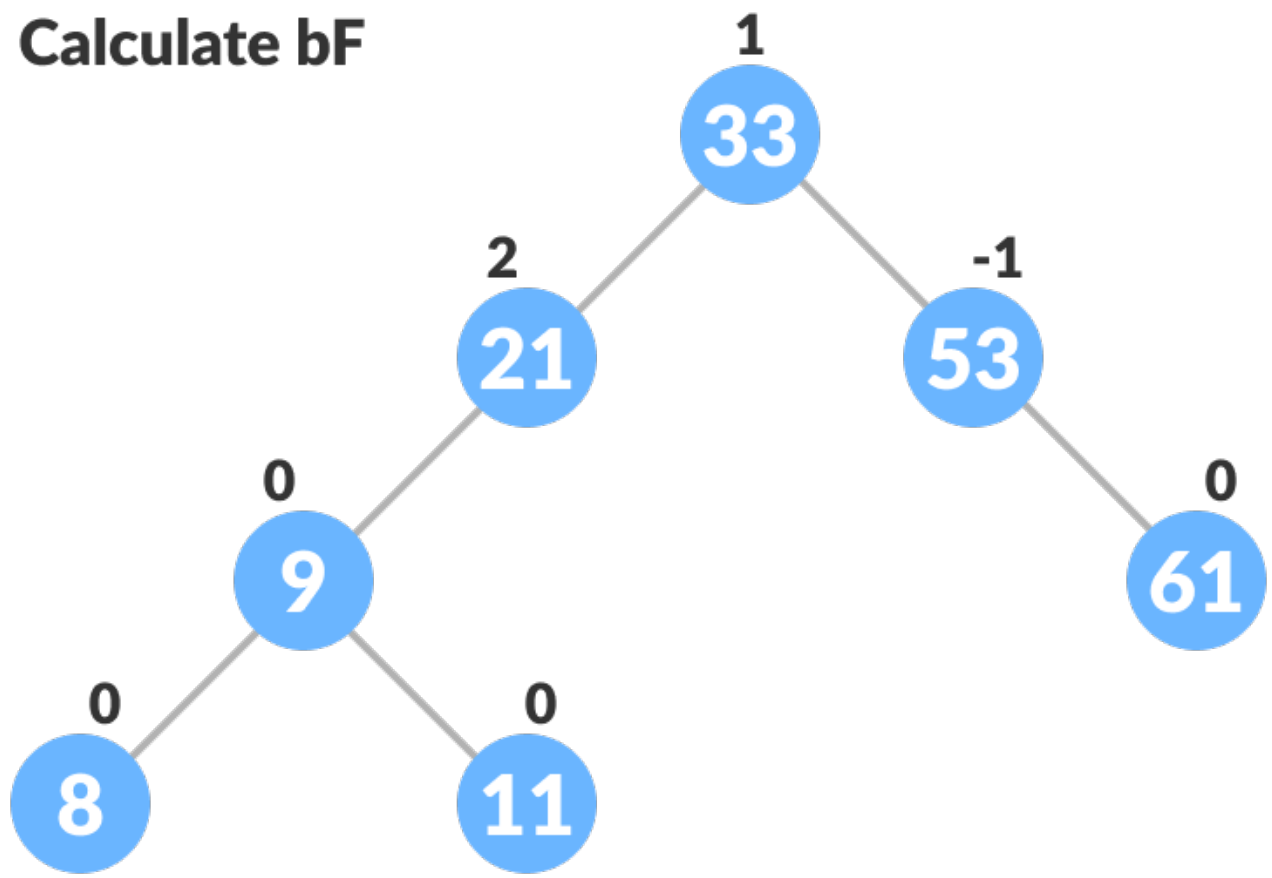
Substitute the node to be deleted



Remove w

Algorithm to Delete a node

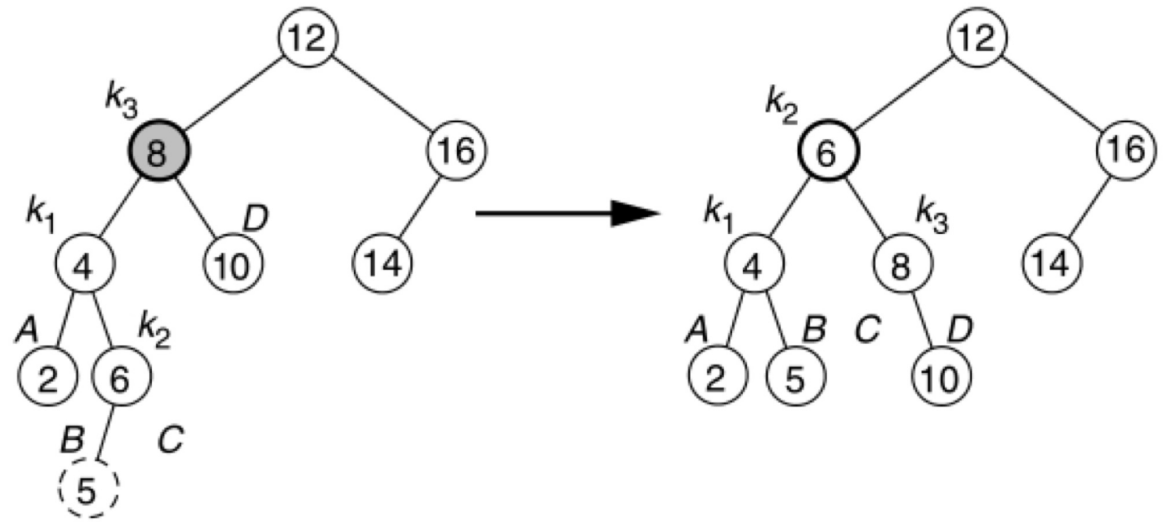
- **Step 4.** Update balanceFactor of the nodes.



Algorithm to Delete a node

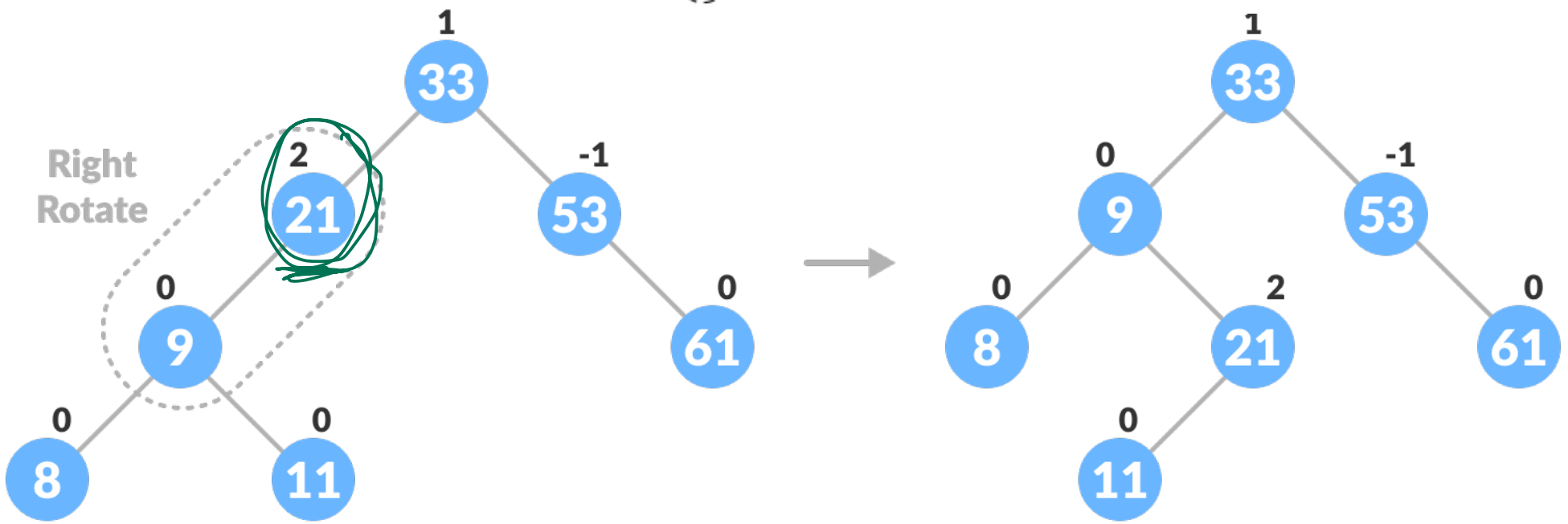
Step 5. Rebalance the t
is not equal to -1, 0 or 1

a) If balanceFacto
   i. If balanceFa
   ii. Else do left-



odes

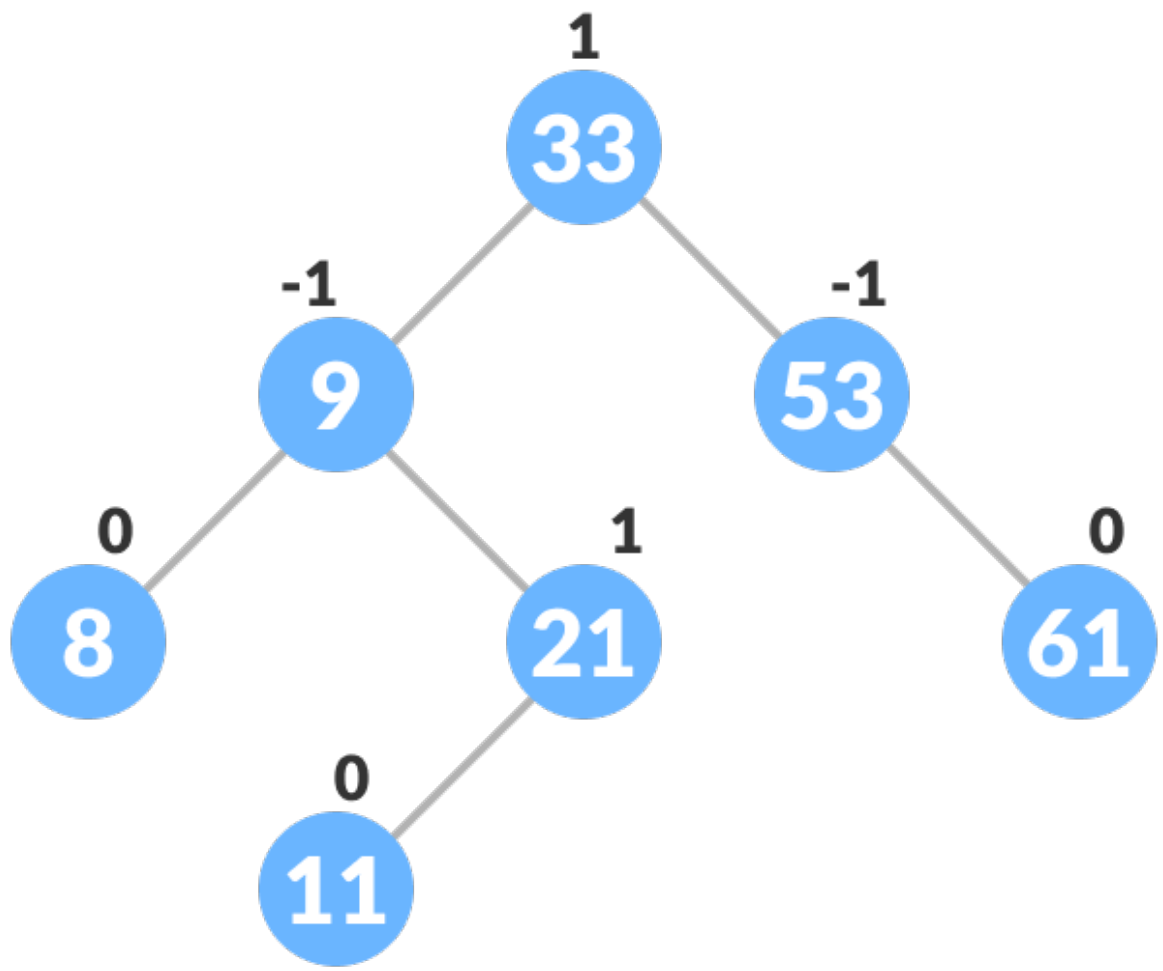
n.



b) If balanceFactor of currentNode < -1,
   i. If balanceFactor of rightChild <= 0, do left rotation.
   ii. Else do right-left rotation.

Algorithm to Delete a node

- The final tree is:



Delete (2/2)

```
// Update the balance factor of each node and balance the tree
        root.height = max(height(root.left), height(root.right)) + 1;
        int balanceFactor = getBalanceFactor(root);
        if (balanceFactor > 1) {
                if (getBalanceFactor(root.left) >= 0) { return rightRotate(root); }
                else {
                        root.left = leftRotate(root.left);
                        return rightRotate(root);
                }
        }
        if (balanceFactor < -1) {
                if (getBalanceFactor(root.right) <= 0) { return leftRotate(root); }
                else {
                        root.right = rightRotate(root.right);
                        return leftRotate(root);
                }
        }
        return root;
}
```

Delete (1/2)

```
// Delete a node
Node deleteNode(Node root, int item) {
// Find the node to be deleted and remove it
        if (root == null) return root;
        if (item < root.item) root.left = deleteNode(root.left, item);
        else if (item > root.item) root.right = deleteNode(root.right, item);
        else {
                if ((root.left == null) || (root.right == null)) {
                        Node temp = null;
                        if (temp == root.left) temp = root.right;
                        else temp = root.left;
                        if (temp == null) { temp = root; root = null; }
                        else root = temp;
                }
                else {
                        Node temp = nodeWithMimumValue(root.right);
                        root.item = temp.item;
                        root.right = deleteNode(root.right, temp.item);
                }
        }
        if (root == null) return root; // Update the balance factor of each node and
balance the tree root.height = max(height(root.left), height(root.right)) + 1; int
```

- AVL Tree
- Rotation
- Insert
- Delete
- **Max and Min**
- Driver code
- Complexities of Different Operations

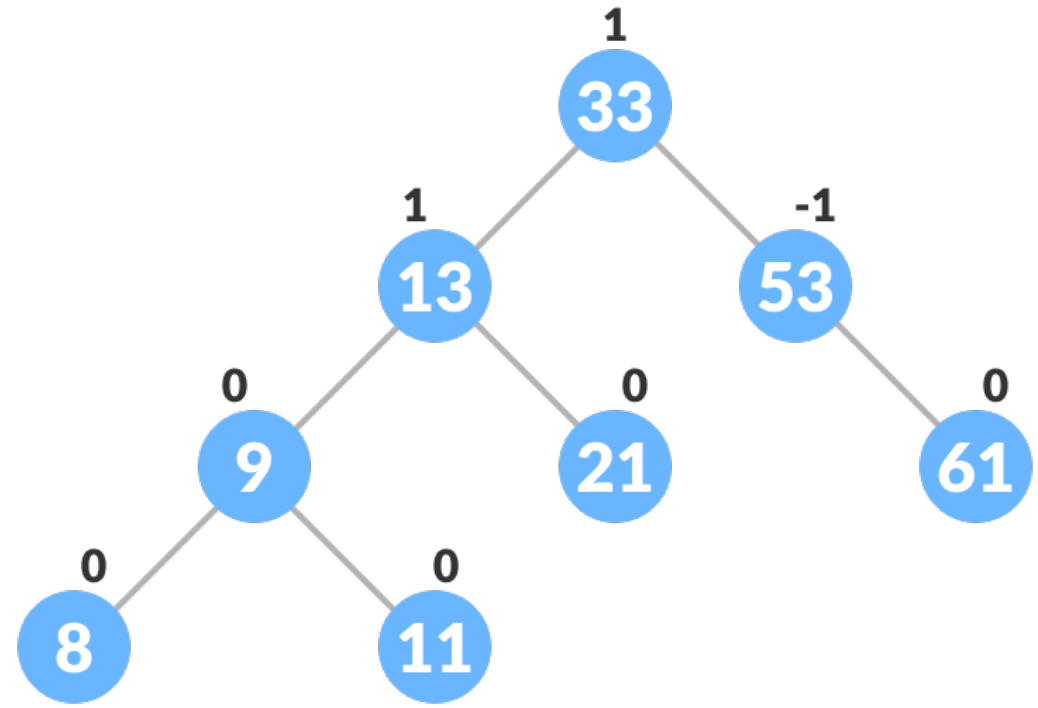
Find Mimum

```
Node nodeWithMimumValue(Node node)
{
    Node current = node;
    while (current.left != null)
        current = current.left;
    return current;
}
```

- AVL Tree
- Rotation
- Insert
- Delete
- Max and Min
- **Driver code**
- Complexities of Different Operations

Driver code

```
// Driver code
public static void main(String[] args) {
        AVLTree tree = new AVLTree();
        tree.root = tree.insertNode(tree.root, 33);
        tree.root = tree.insertNode(tree.root, 13);
        tree.root = tree.insertNode(tree.root, 53);
        tree.root = tree.insertNode(tree.root, 9);
        tree.root = tree.insertNode(tree.root, 21);
        tree.root = tree.insertNode(tree.root, 61);
        tree.root = tree.insertNode(tree.root, 8);
        tree.root = tree.insertNode(tree.root, 11);
        tree.printTree(tree.root, "", true);
        tree.root = tree.deleteNode(tree.root, 13);
        System.out.println("After Deletion: ");
        tree.printTree(tree.root, "", true);
}
```



Print the Tree

```
// Print the tree
private void printTree(Node currPtr, String indent, boolean last) {
      if (currPtr != null) {
            System.out.print(indent);
            if (last) { System.out.print("R----");
            indent += "  ";
      }
      else {
            System.out.print("L----");
            indent += "|  ";
      }
      System.out.println(currPtr.item);
      printTree(currPtr.left, indent, false);
      printTree(currPtr.right, indent, true);
}
```

- AVL Tree
- Rotation
- Insert
- Delete
- Max and Min
- Driver code
- **Complexities of Different Operations**

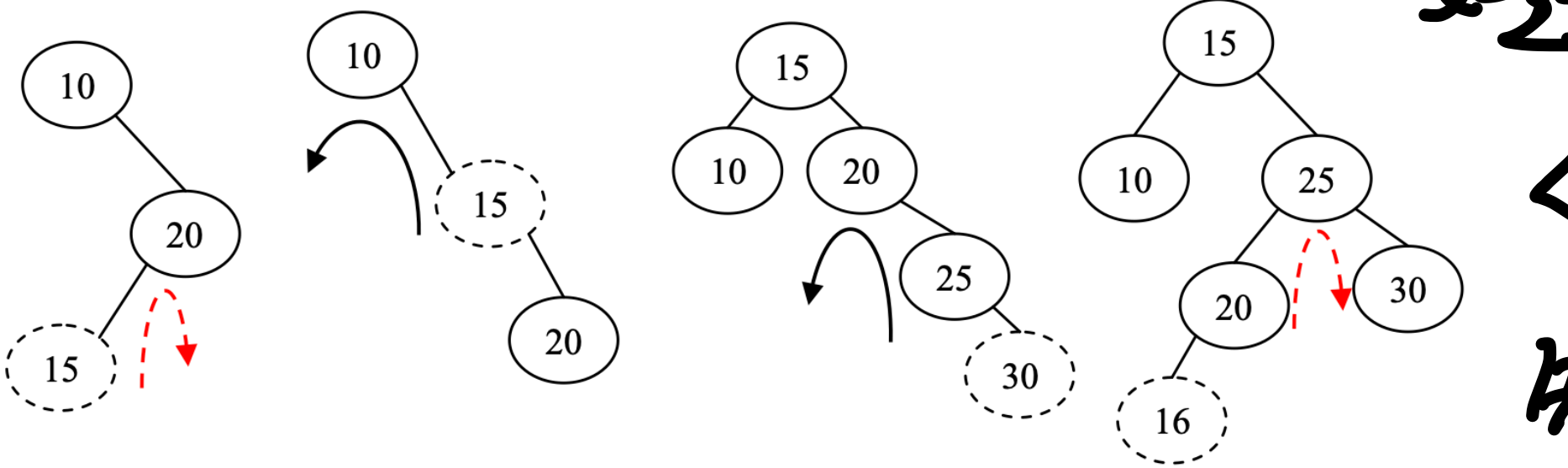
Complexities of Different Operations

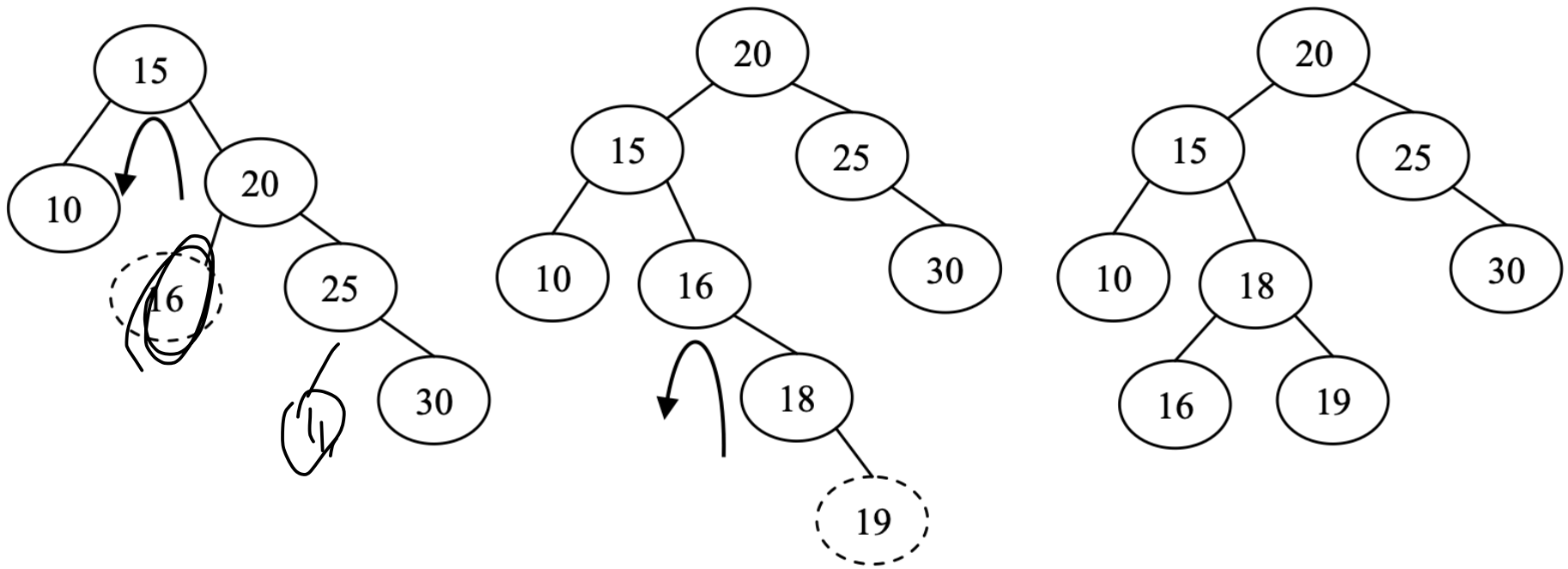
Insertion	Deletion	Search
O(log n)	O(log n)	O(log n)

Exercise△ 从下往上第一个不平衡的点，

- Insert the following sequence of elements into an AVL tree, starting with an empty tree: **10, 20, 15, 25, 30, 16, 18, 19**.

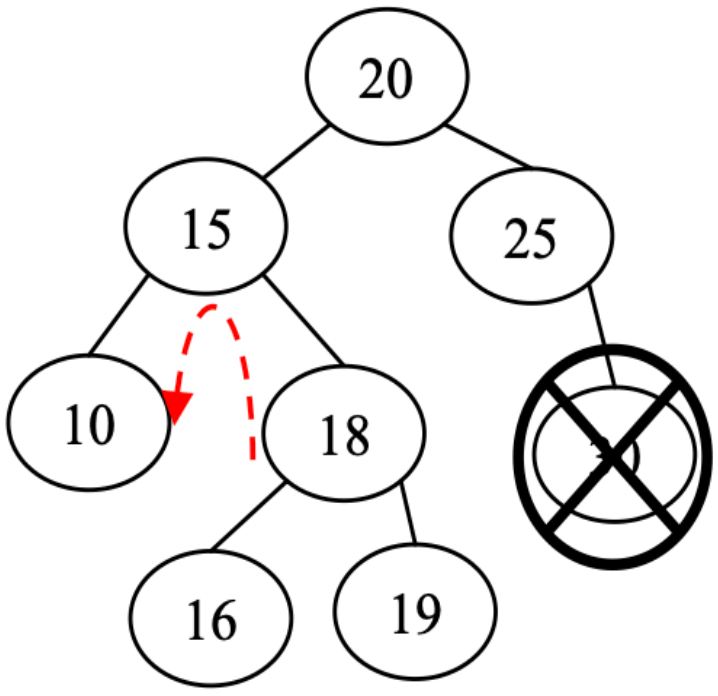
如遇一个，先变直，

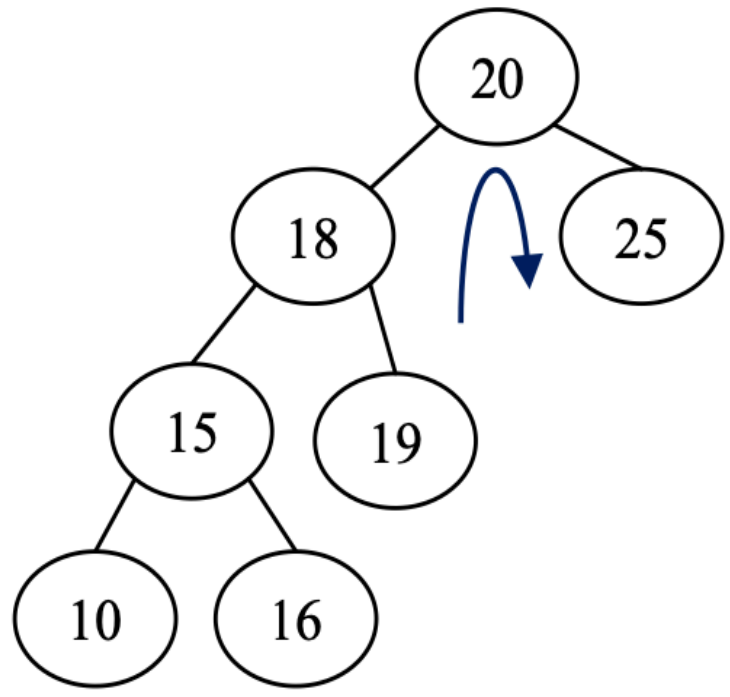


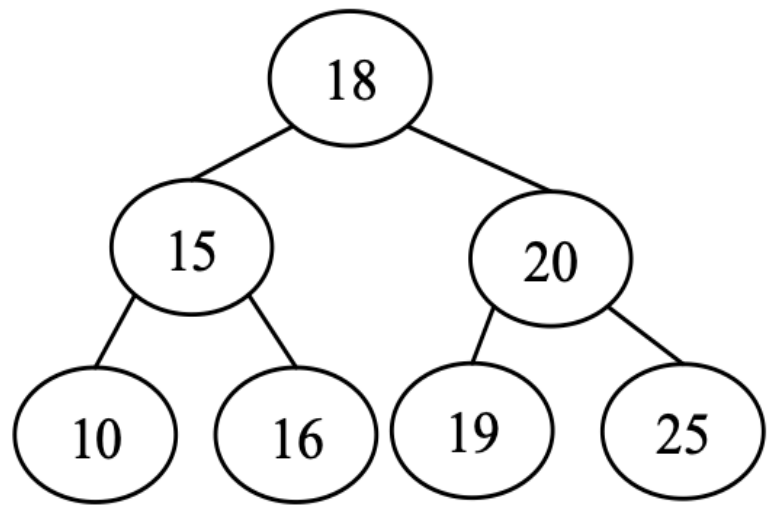


Exercise

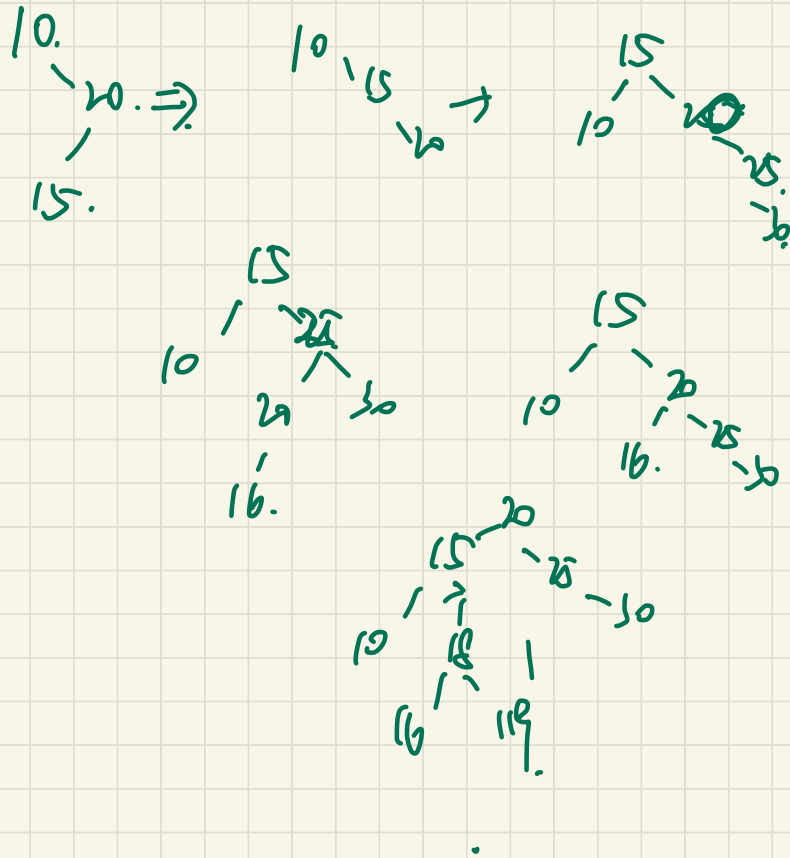
- Delete 30 in the AVL tree that you got.





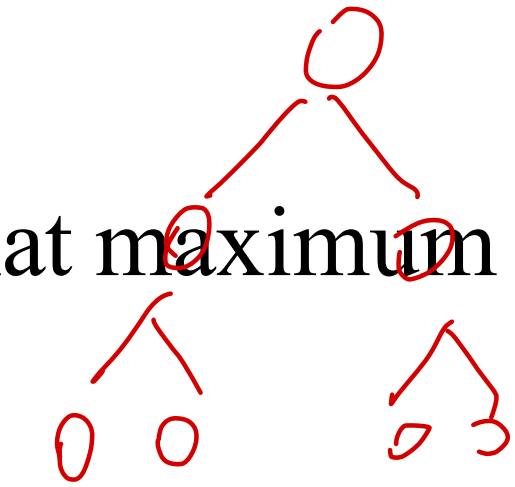


10, 20, 15, 25, 30, 16, 18, 19.

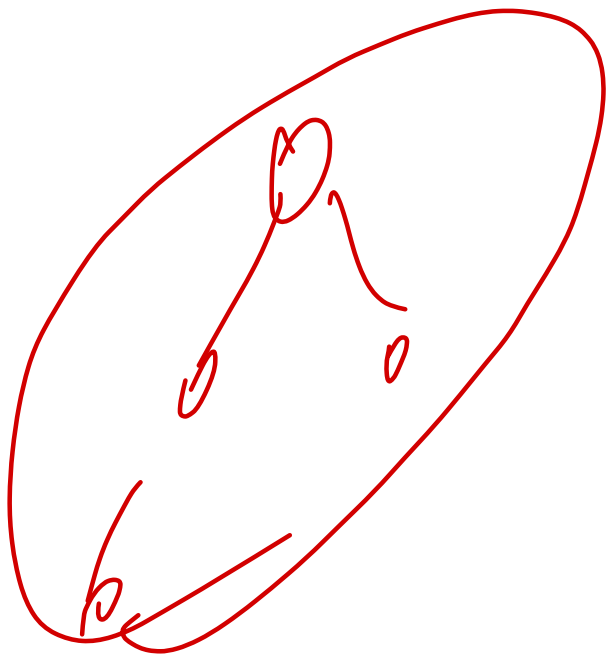


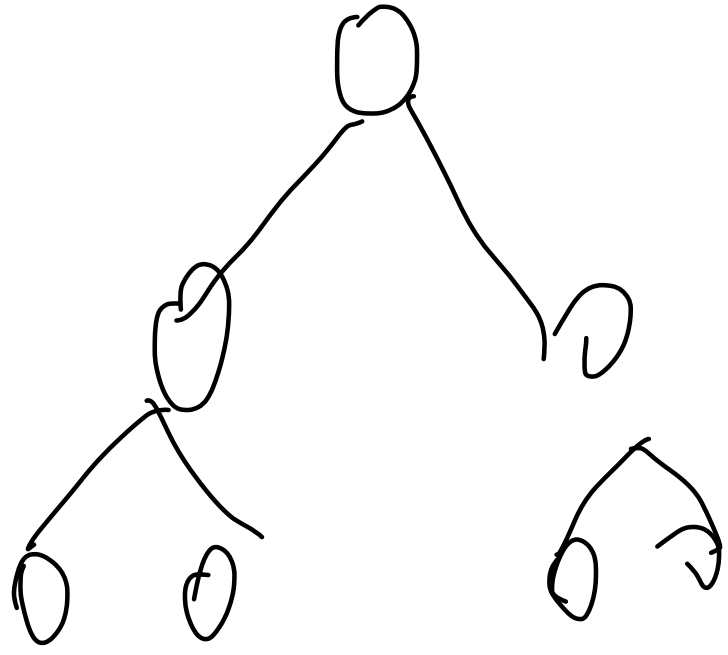
Exercise

- If an AVL tree has a height of 3, what maximum number of nodes can the tree have?
  - Ans: 7



- What minimum number of nodes can the tree have?
  - Ans: 4





Exercise: AVL Properties Give log(N) height

- Lemma (little theorem) (Thm 19.3 in Weiss, pg 708, adapted)
  - An AVL Tree of height H has at least $F_{H+2} - 1$ nodes where Fi is the i-th Fibonacci number.

- Definitions:
  - Fi: i-th Fibonacci number (0,1,1,2,3,5,8,13,...)
  - $F_i = F_{i-1} + F_{i-2}$ for $i > 1$
  - S:sizeofatree
  - H: height (assume roots have height 1)
  - S_H is the smallest size AVL Tree with height H

Exercise: AVL Properties Give log(N) height

- An AVL Tree of height H has at least $F_{H+2} - 1$ nodes where Fi is the i-th Fibonacci number.
- Proof.
  - Proof by Induction: Base Cases True

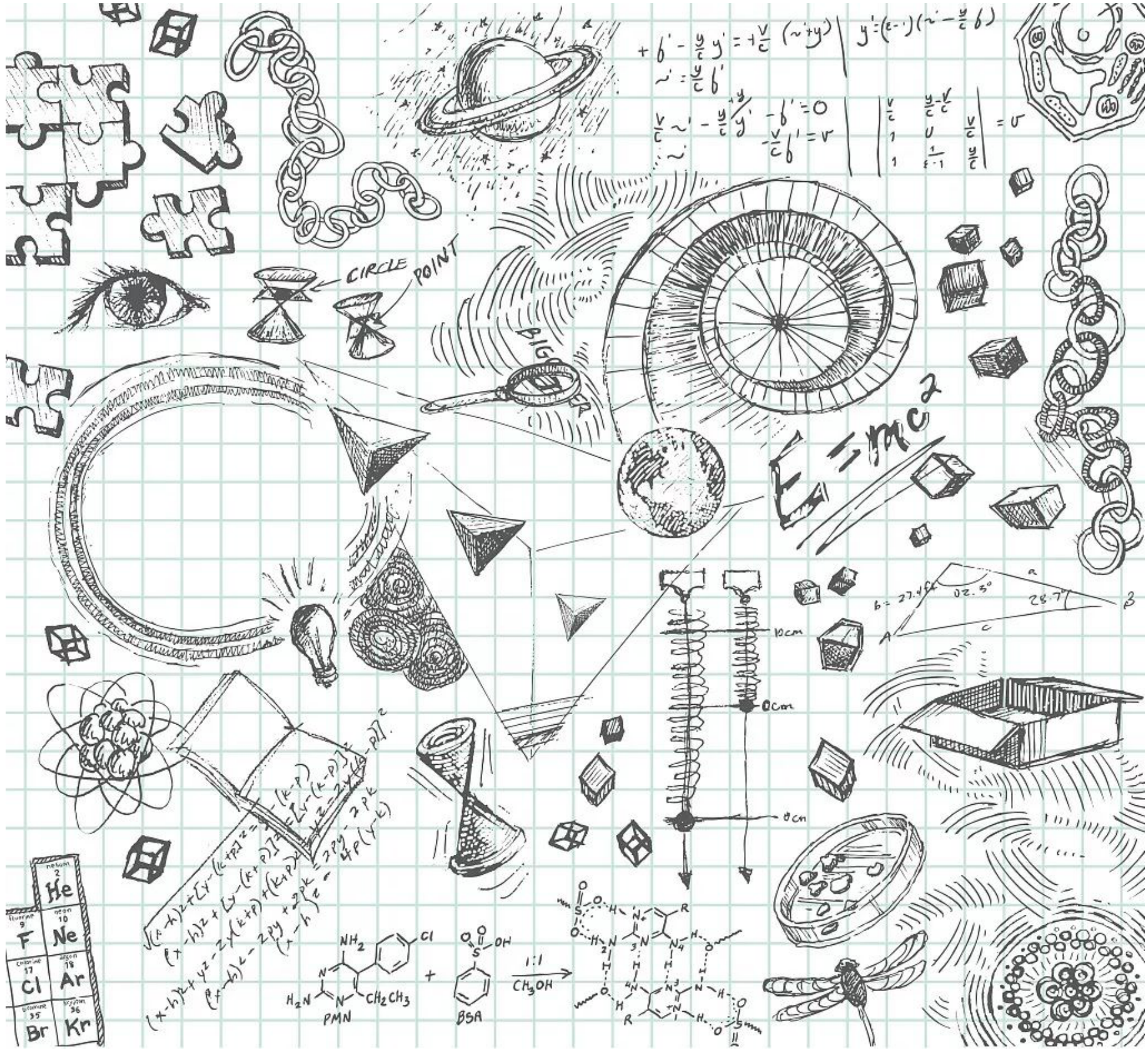
Tree	height	Min Size	Calculation
empty	$H = 0$	S_0	$F_{(0+2)} - 1 = 1 - 1 = 0$
root	$H = 1$	S_1	$F_{(1+2)} - 1 = 2 - 1 = 1$
root+(left or right)	$H = 2$	S_2	$F_{(2+2)} - 1 = 3 - 1 = 2$

Exercise: AVL Properties Give log(N) height

- An AVL Tree of height H has at least $F_{H+2} - 1$ nodes where Fi is the i-th Fibonacci number.
- Proof. (conti.)
  - Consider an arbitrary AVL tree T with height H > 2
  - Let S_H smallest size for tree T
  - Assume equation true for smaller trees
    - Notice: Left/Right are smaller AVL trees
    - Notice: Left/Right differ in height by at most 1

Exercise: AVL Properties Give log(N) height

- Proof. (conti.)
  - T has height H
  - Assume for height h < H, smallest size of T is $S_h = F_{h+2} - 1$
  - Suppose **Left is 1 higher than Right**
  - LeftHeight: h=H−1
  - Left Size: $F_{(H-1)+2} - 1 = F_{H+1} - 1$
  - RightHeight: h=H−2
  - Right Size: $F_{(H-2)+2} - 1 = F_H - 1$
  - S_H = size(Left)+size(Right)+1
    $=(F_{H+1} - 1)+(F_H - 1)+1$
    $= F_{H+1} + F_H - 1$
    $=F_{H+2}-1$


CIRCLE
POINT
BIG
E=mc2
10cm
0cm
8cm
PMN
BSA
1:1
CH3OH

AVL:

$$F_{h+2} - 1 \leq node \leq 2^{h-1}$$