AVL Tree

Joy Mukherjee

Assistant Professor
Computer Science & Engineering
IIT Bhubaneswar

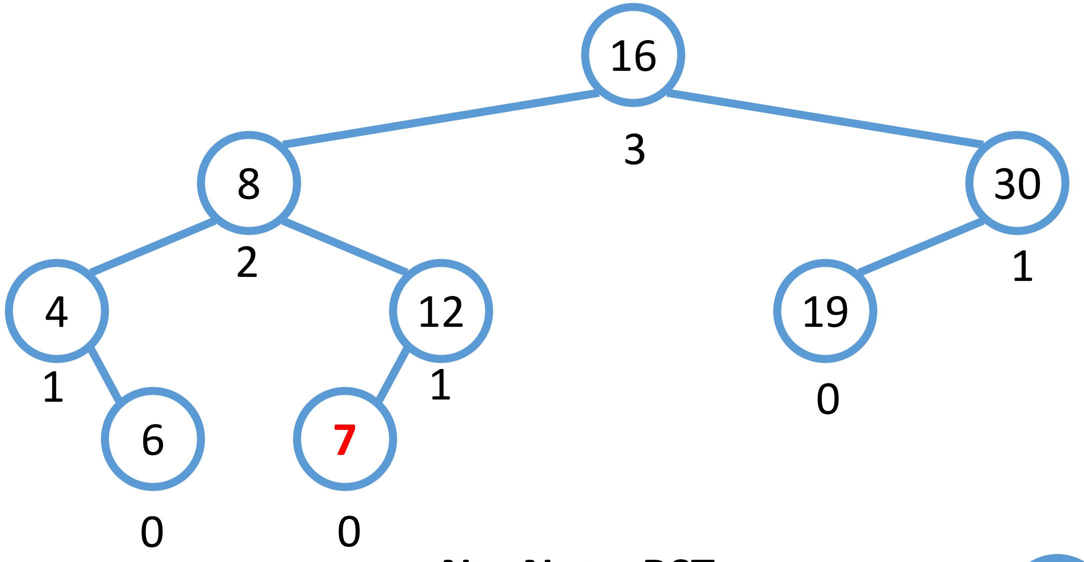
AVL Tree / Height Balanced Tree

1. **An AVL tree is a binary search tree.**
2. **For every internal node v, the heights of the children of v can differ by at most one.**

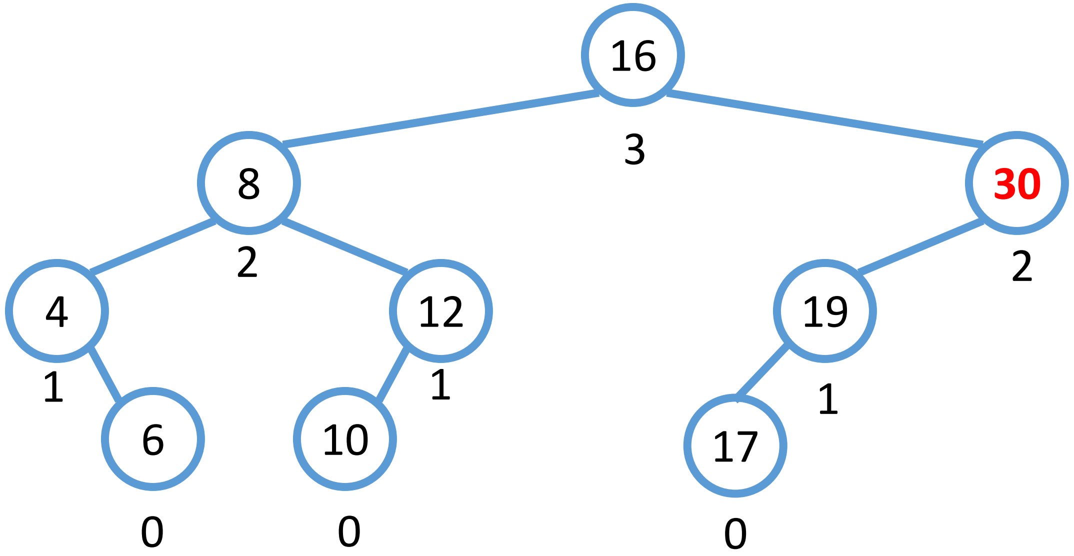
```
typedef struct node {
        int data;
        struct node *left;
        struct node *right;
        int height;
}node;
```

Balance Factor = Height of the left subtree – height of the right subtree
In an AVL tree, |Balance Factor| <= 1

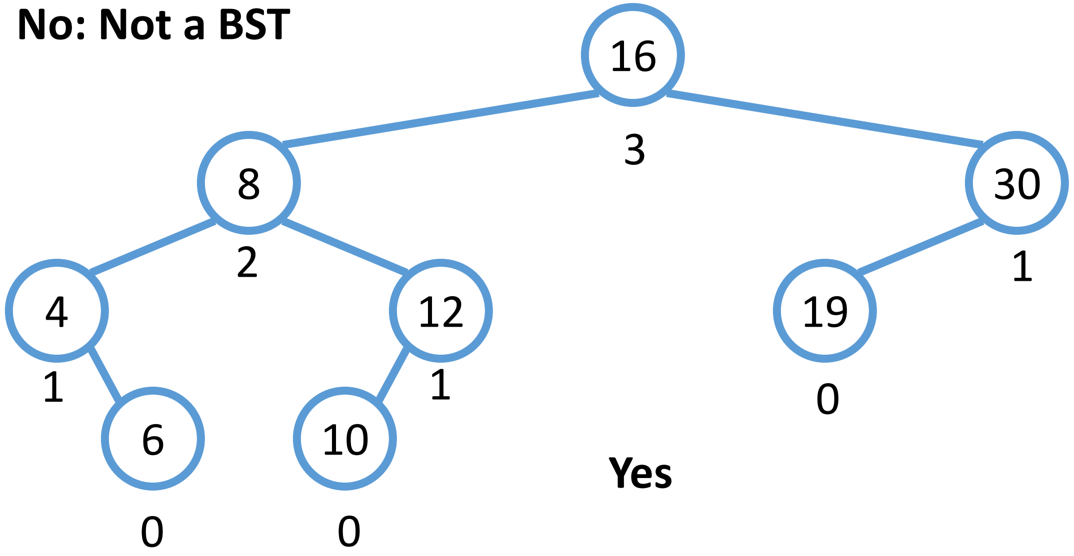
AVL Tree vs Non-AVL Tree



No: Not a BST



No: Height Imbalance



Yes

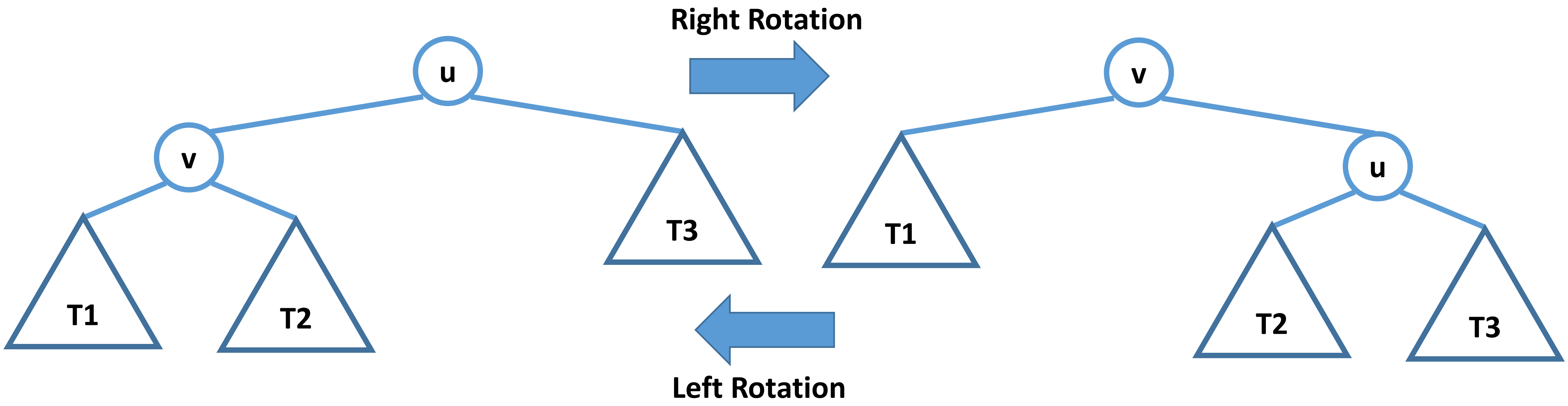
Height of an AVL Tree

Height & Balance Factor of a Node

```
int findHeight(node *root)
{
   if(root == NULL) return -1;
   int hLeft = findHeight(root->left);
   int hRight = findHeight(root->right);
   return (hLeft >= hRight) ? hLeft+1 : hRight+1;
}
int getBalanceFactor(node *root)
{
   return findHeight(root->left) - findHeight(root->right);
}
```

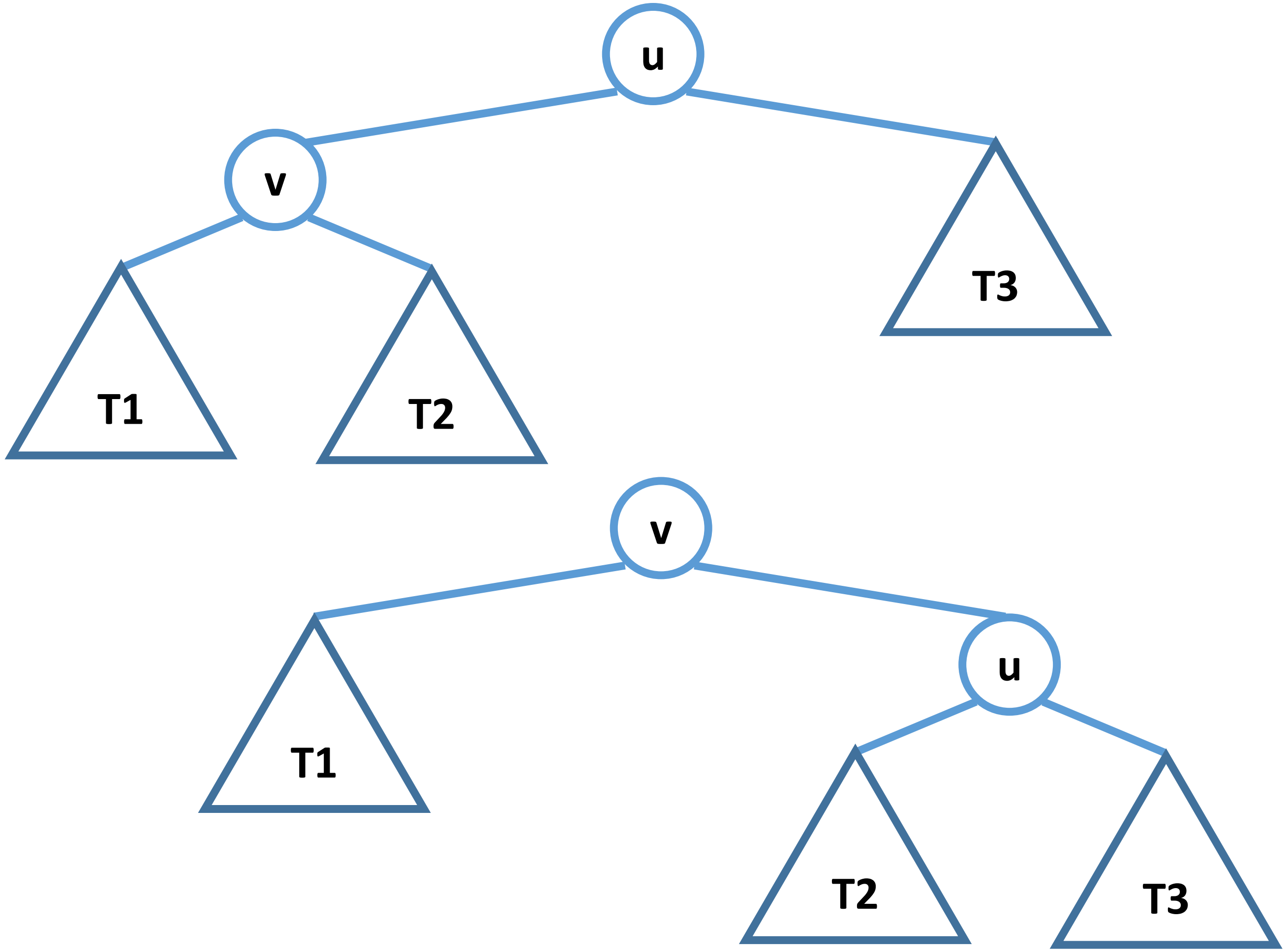
Rotation

- Rotation is an operation that reorganizes locally a BST to retain the height-balanced property.
- Let u and v be two nodes such that u is a parent of v. The left and right subtrees of v are T1 and T2 respectively. T3 is the right subtree of u.



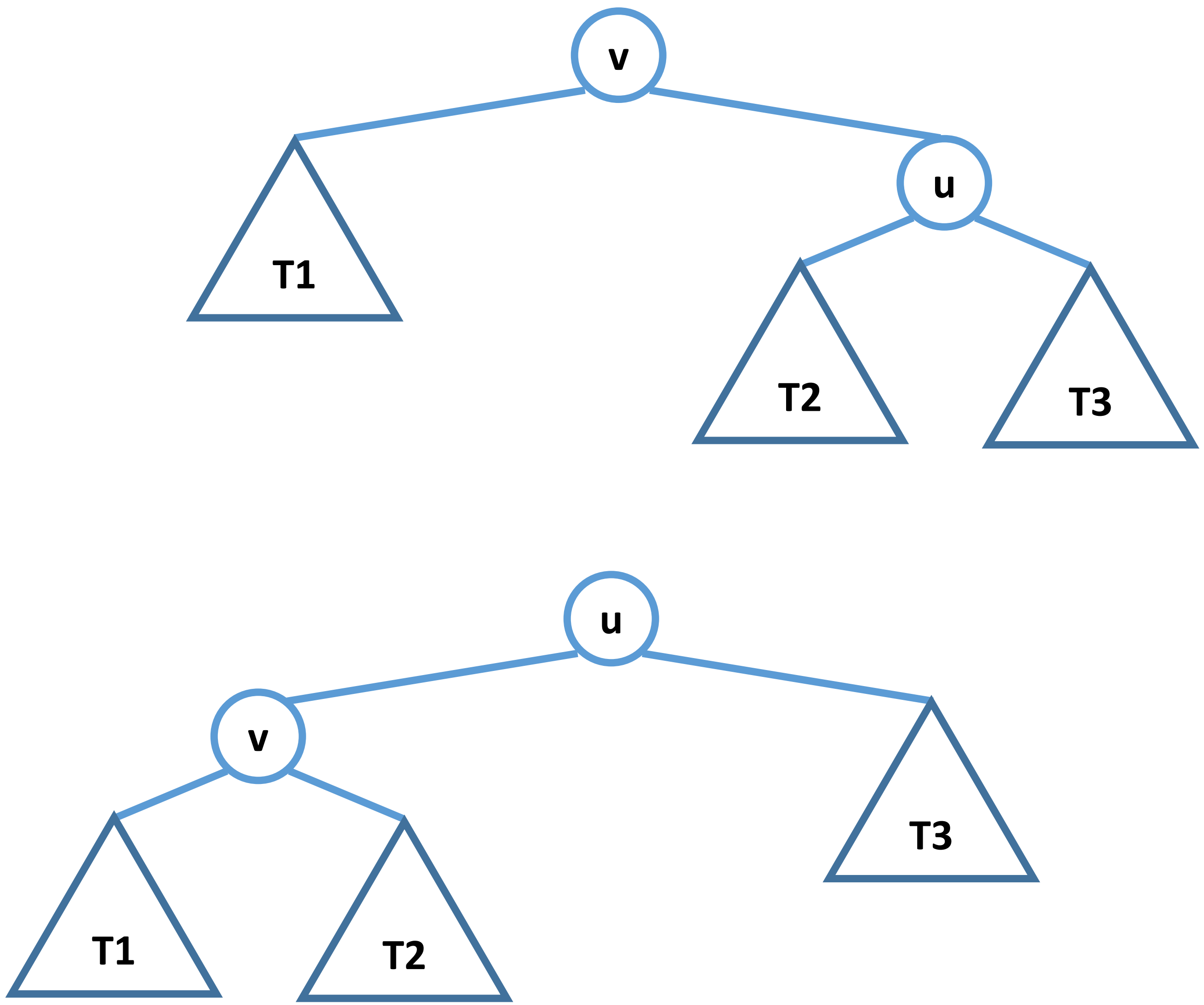
Right Rotation

```
node *rightRotate(node *u)
{
   node *v = u->left;
   u->left = v->right;
   v->right = u;
   u->height = findHeight(u);
   v->height = findHeight(v);
   return v;
}
```



Left Rotation

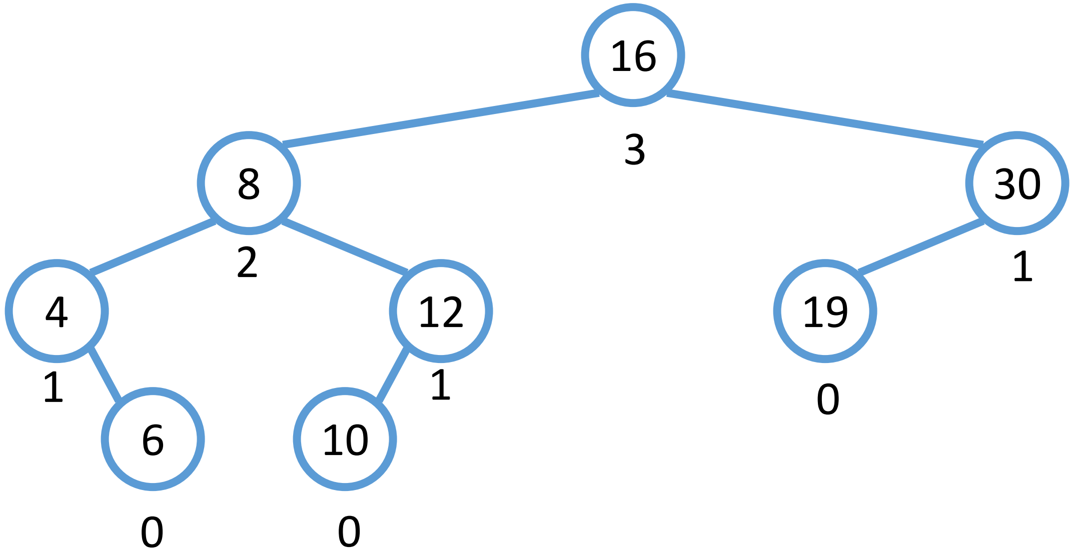
```
node *leftRotate(node *v)
{
   node *u = v->right;
    v->right = u->left;
   u->left = v;
   v->height = findHeight(v);
   u->height = findHeight(u);
   return u;
}
```

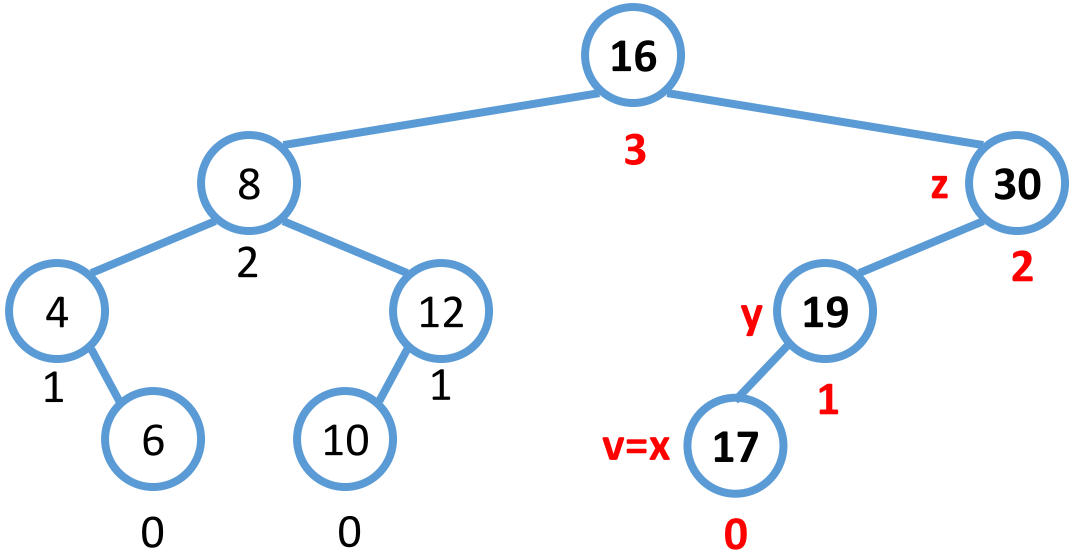


Insertion in AVL Tree

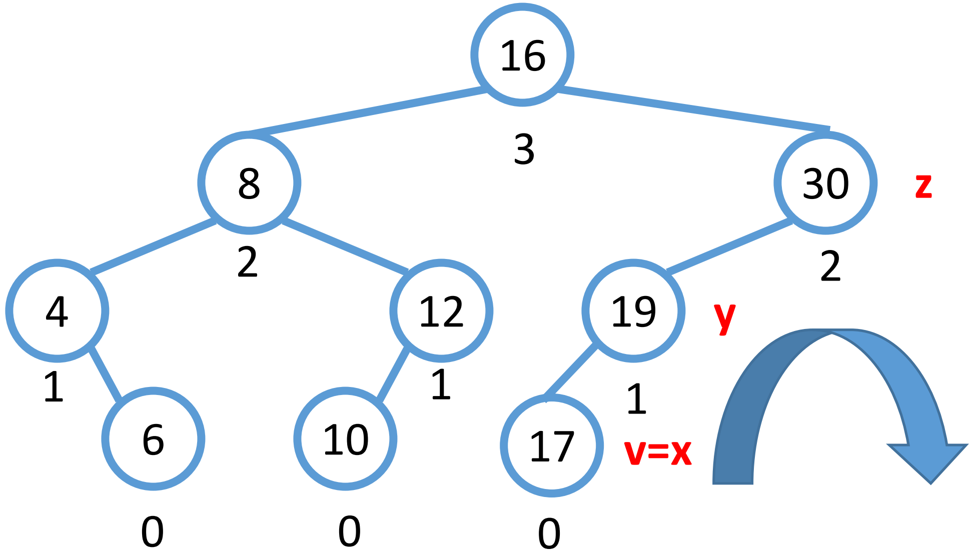
Effect of Insertion

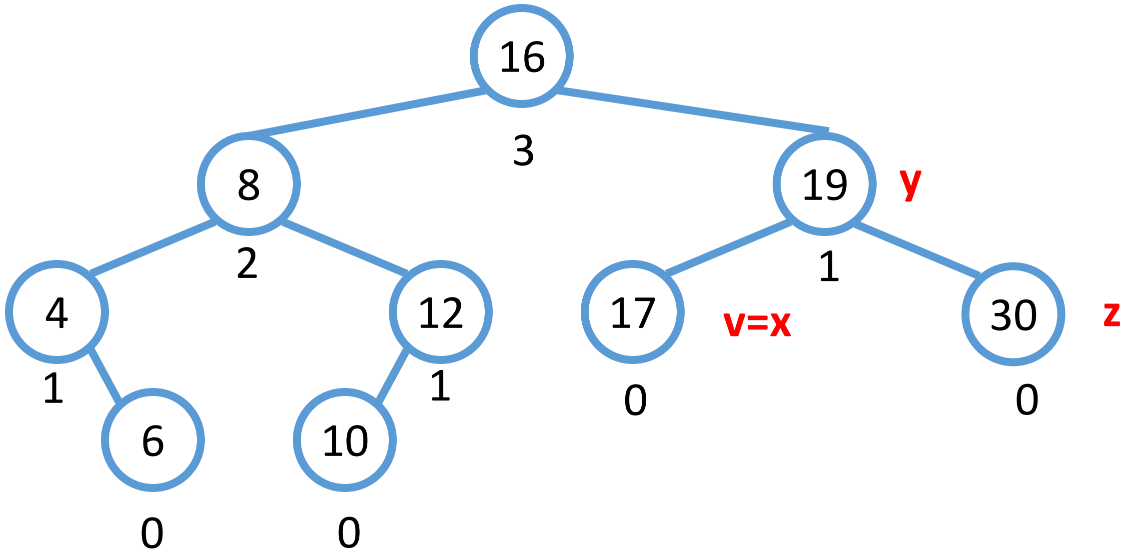
- Insert node v (=17)
- Heights of the ancestors of v can only increase.
- If insertion causes height imbalance, then we travel up the tree from v until we find the first node x such that its grandparent z is not height balanced.
- Let, y be the parent of x and child of z.
- To rebalance the subtree rooted at z, we must perform a rotation.





Insertion ➡ Height Imbalance ➡ Rotation





Types of Rotation

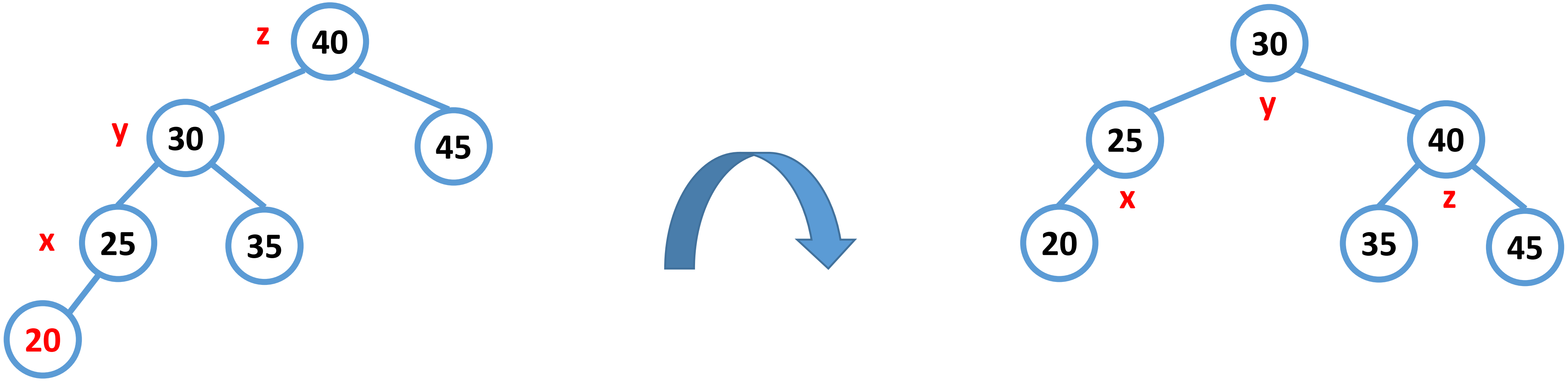
1. Right Rotation
2. Left-Right Rotation
3. Left Rotation
4. Right-Left Rotation

Insertion

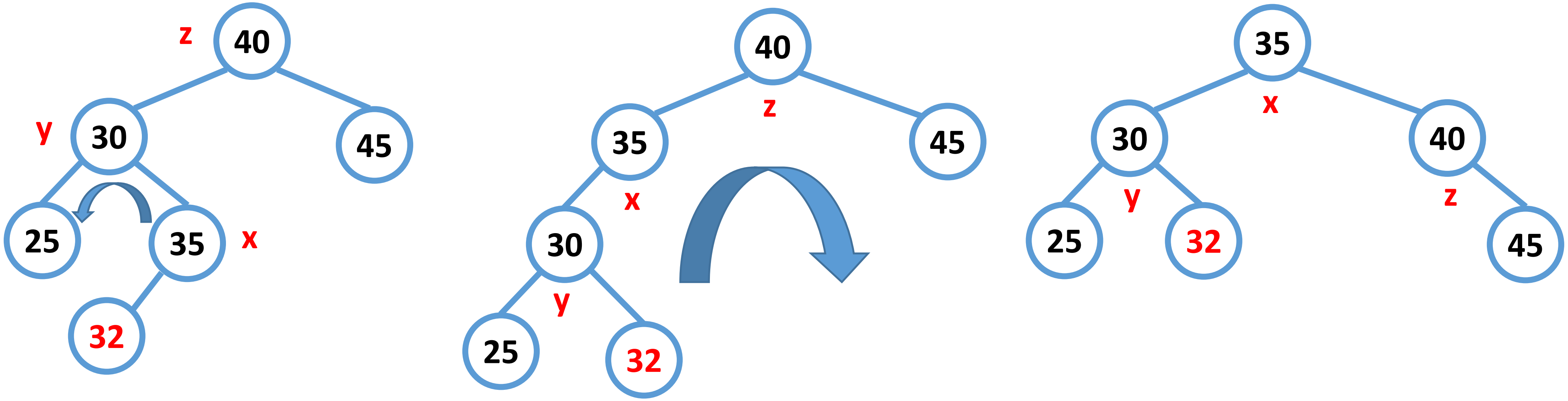
- Imbalance happens at the node (root) where the balance factor is +2 or -2.
- Balance factor   Rotation
- -2, -1   left(root)
- +2, +1   right(root)
- +2, -1   left(left), right(root)
- -2, +1   right(right) left(root)

- If I find any imbalance, I have four cases. If I solve the proper case, I am done with my insertion, and no more rotation is needed.

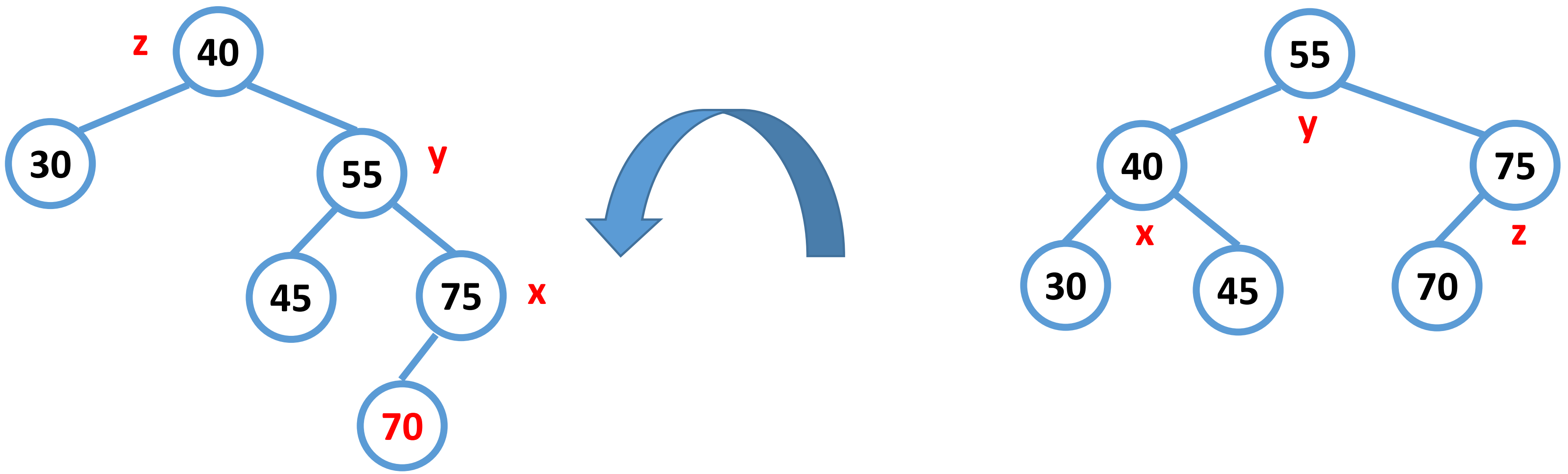
Right Rotation



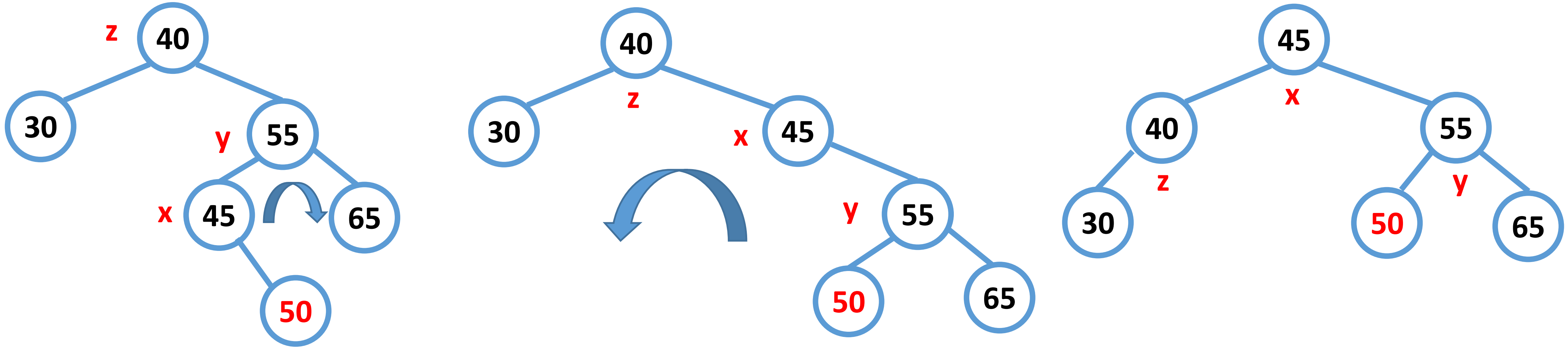
Left-Right Rotation



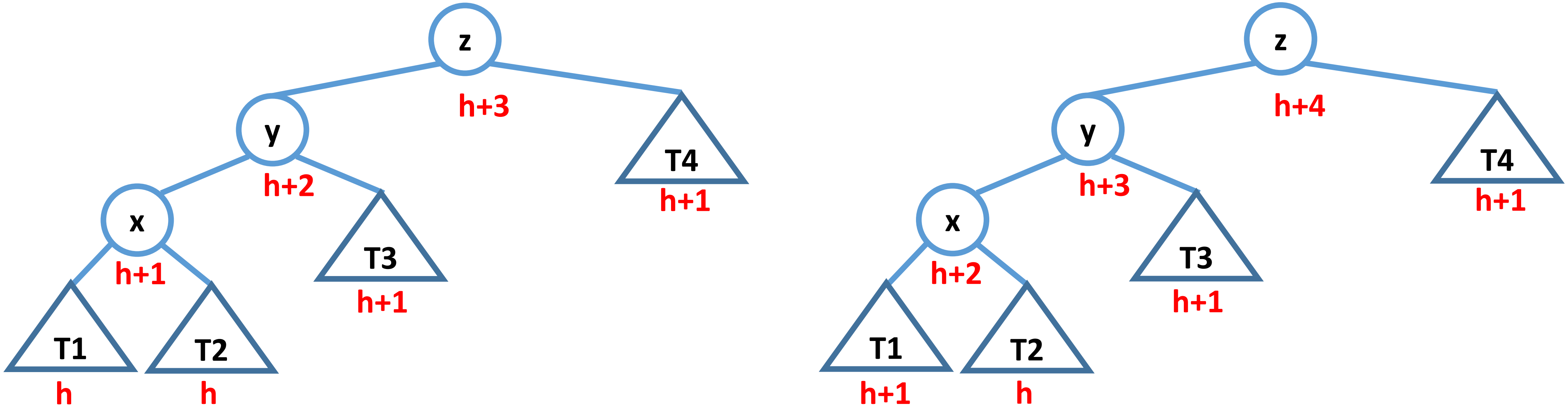
Left Rotation



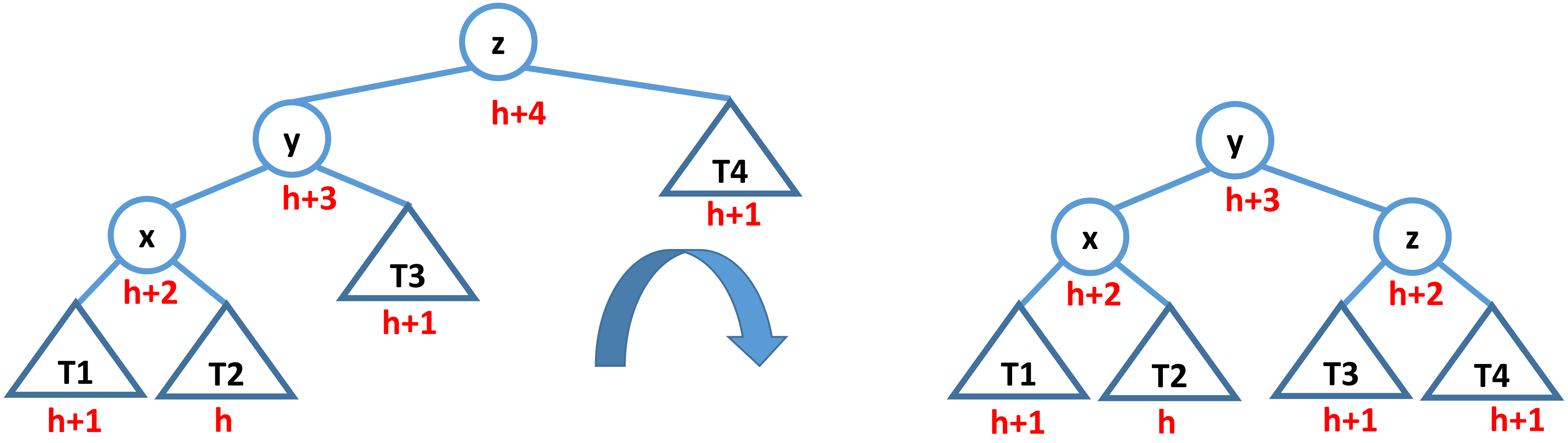
Right-Left Rotation



Single Rotation: Insertion at T1

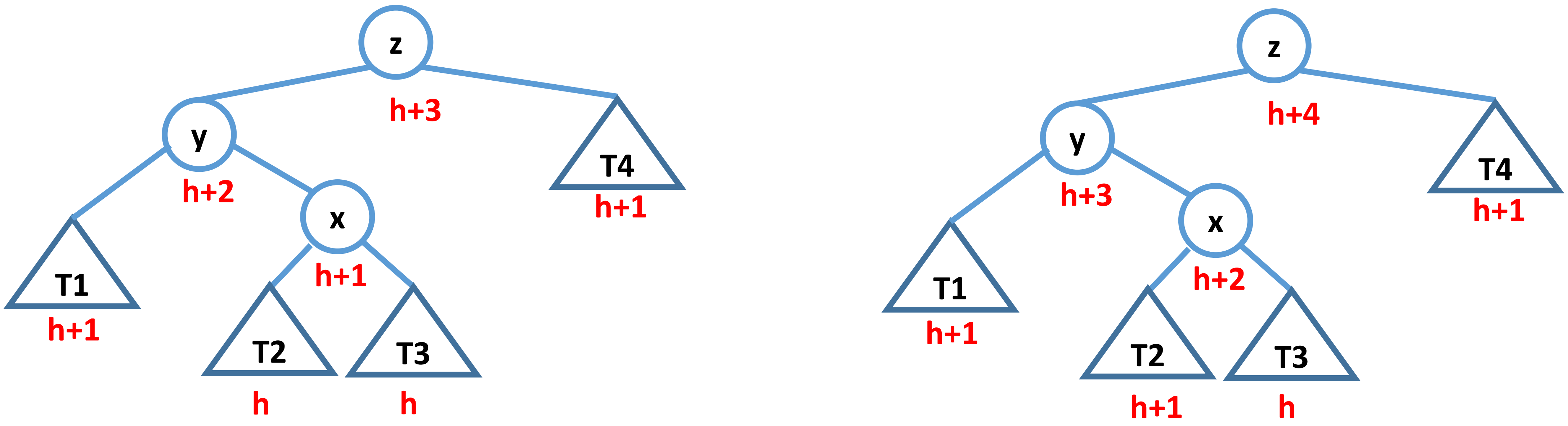


Single Rotation: Insertion at T1

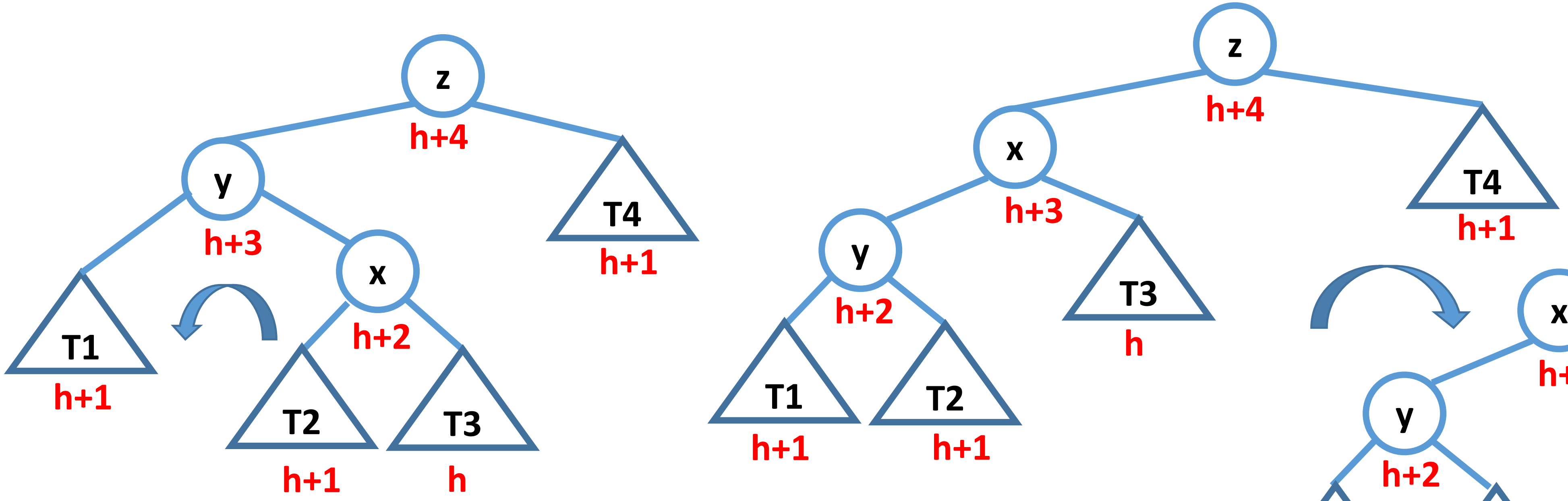


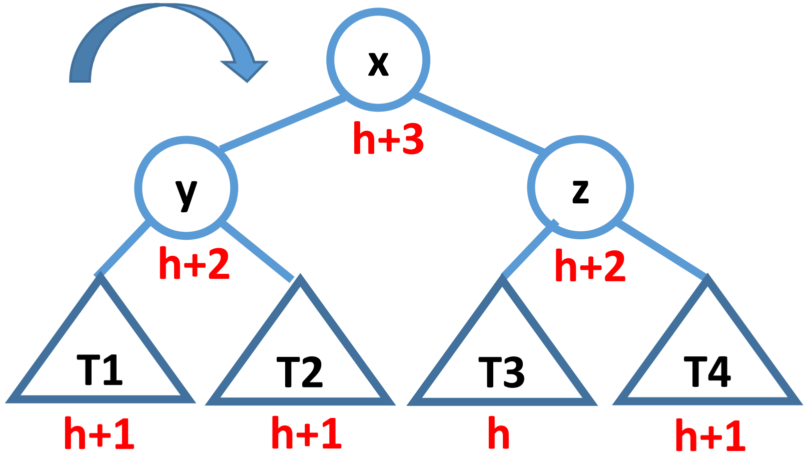
Height of final tree = Height of original tree ➡ No further rotation

Double Rotation: Insertion at T2



Double Rotation: Insertion at T2





Height of final tree = Height of original tree

No further rotation

Code for Insertion into an AVL Tree

```
node *insert(node *root, int data)
{
   if(root == NULL)                root = createNode(data);
   else if(data < root->data)      root->left = insert(root->left, data);
   else if(data > root->data)      root->right = insert(root->right, data);

   root->height = findHeight(root);

   int balanceFactorRoot = getBalanceFactor(root);
```

Left is Heavier than Right

```
if(balanceFactorRoot == 2) {
        int balanceFactorLeft = getBalanceFactor(root->left);
        if(balanceFactorLeft == 1) {
                root = rightRotate(root);
        }
        else if(balanceFactorLeft == -1) {
                root->left = leftRotate(root->left);
                root = rightRotate(root);
        }
  }
```

Right is Heavier than Left

```
    else if(balanceFactorRoot == -2) {
            int balanceFactorRight = getBalanceFactor(root->right);
            if(balanceFactorRight == -1) {
                    root = leftRotate(root);
            }
            else if(balanceFactorRight == 1) {
                    root->right = rightRotate(root->right);
                    root = leftRotate(root);
            }
    }
    return root;
}
```

Deletion in AVL Tree

Types of Rotation

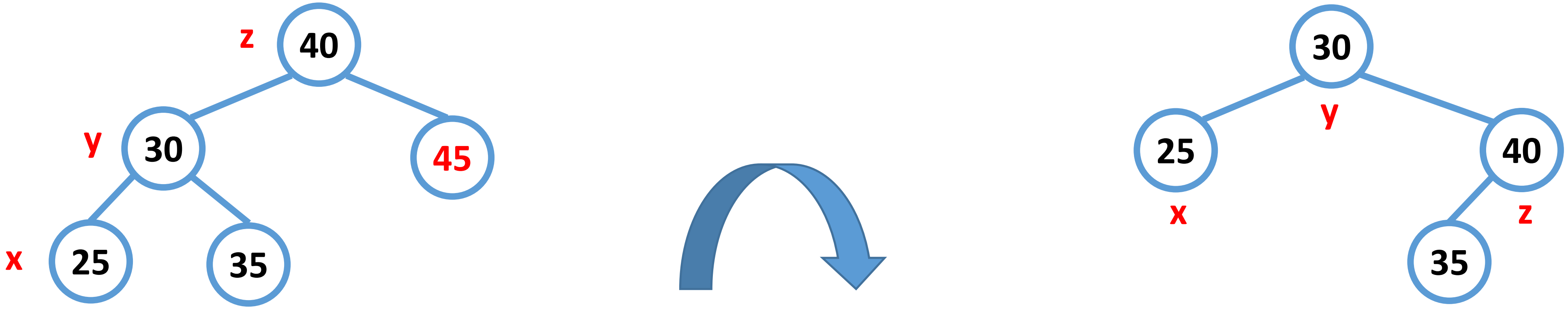
1. Right Rotation
2. Left-Right Rotation
3. Left Rotation
4. Right-Left Rotation

Deletion

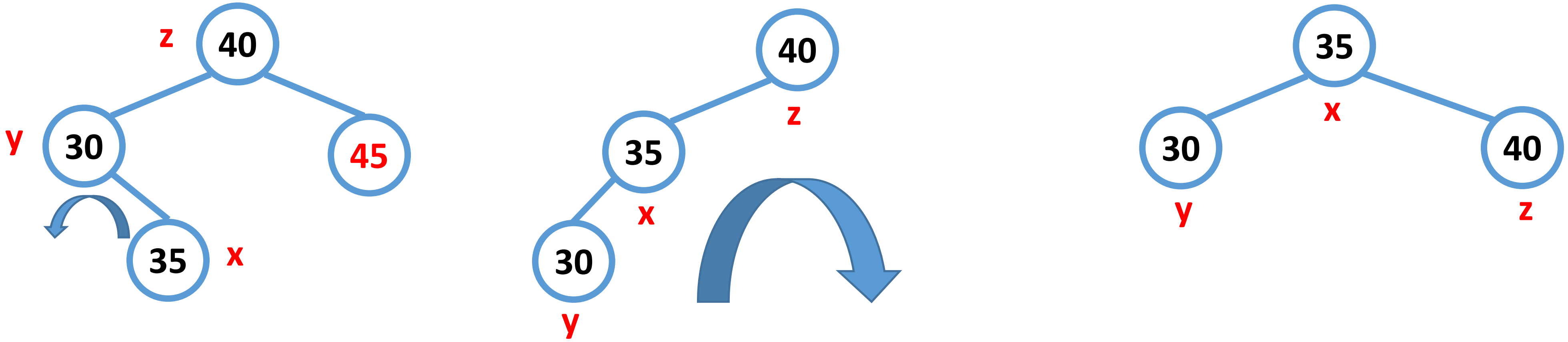
Balance factor	Rotation
-2, -1/0	left(root)
+2, +1/0	right(root)
+2, -1	left(left), right(root)
-2, +1	right(right) left(root)

- If I find any imbalance, I have four cases. If I solve the proper case, I am done with my insertion, and no more rotation is needed.

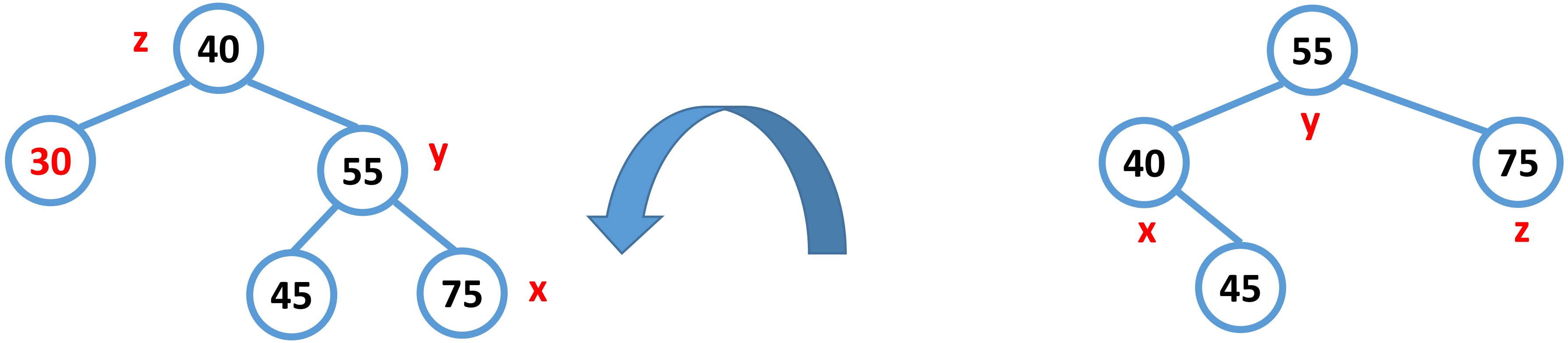
Right Rotation



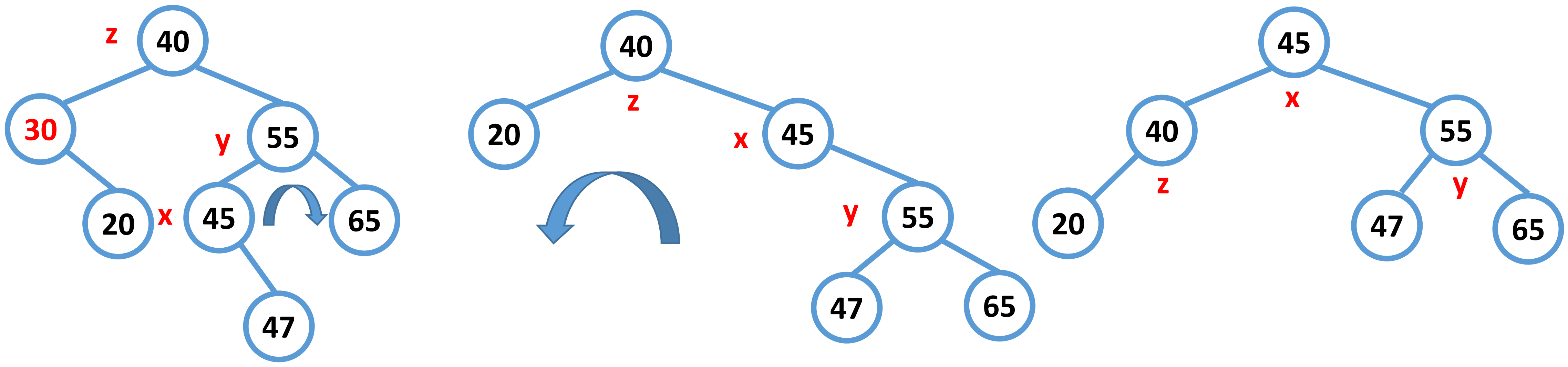
Left-Right Rotation



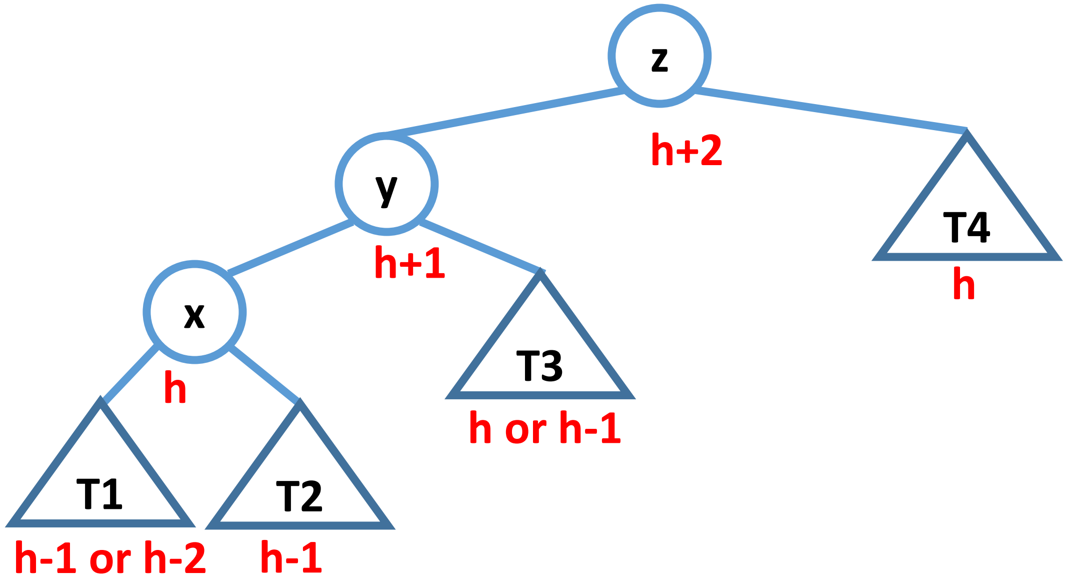
Left Rotation

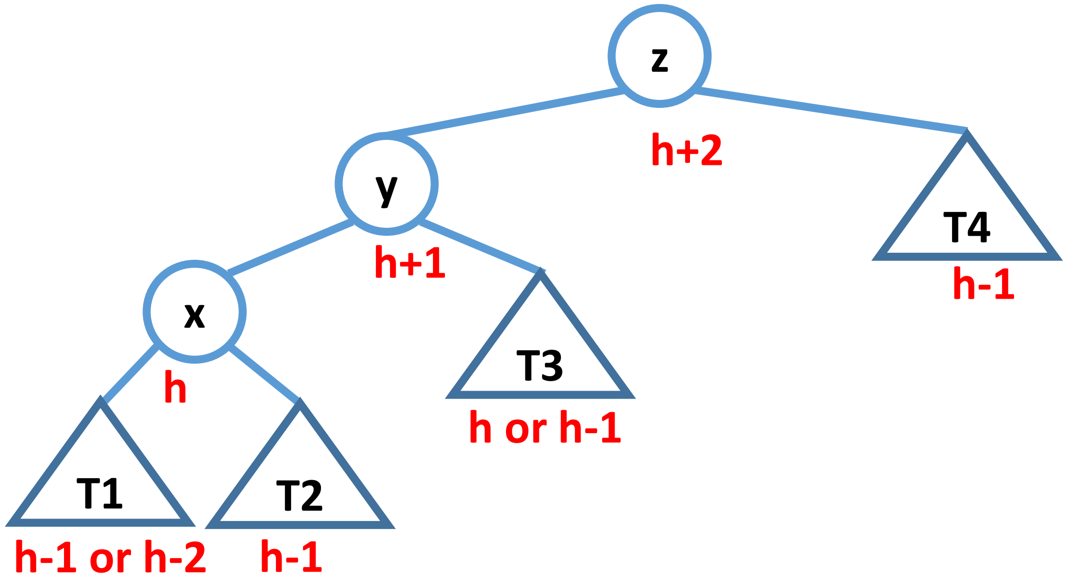


Right-Left Rotation



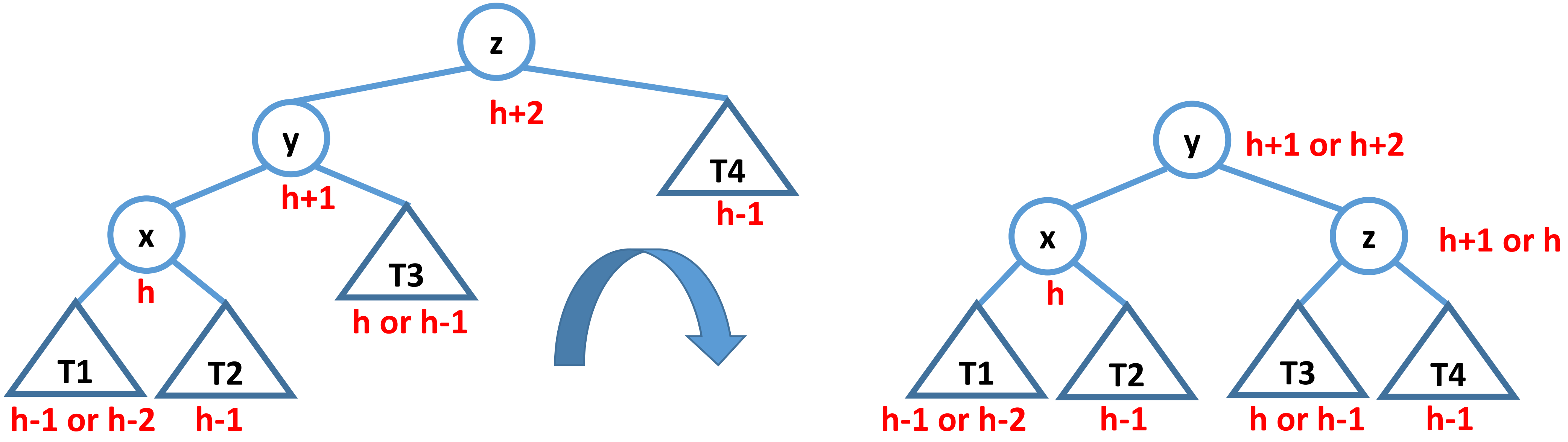
Single Rotation: Deletion at T4





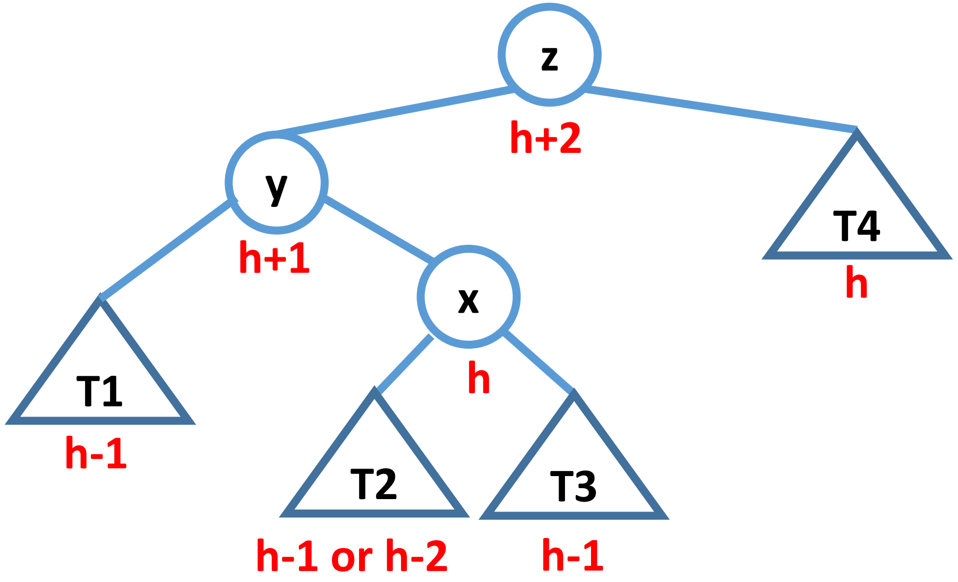
Height of T1 or T2 must be h-1

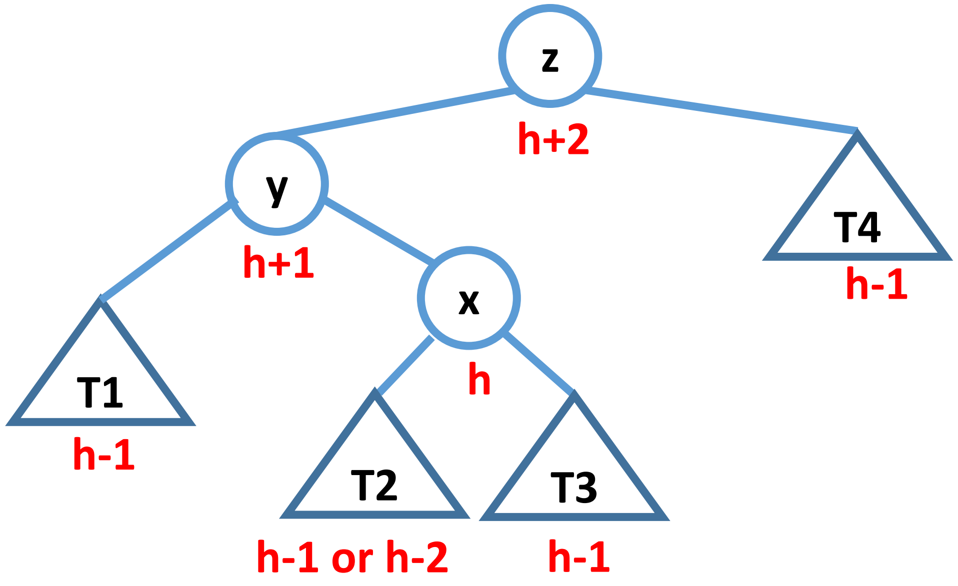
Single Rotation: Deletion at T4



Height of final tree may decrease ➡ Check for height imbalance upwards

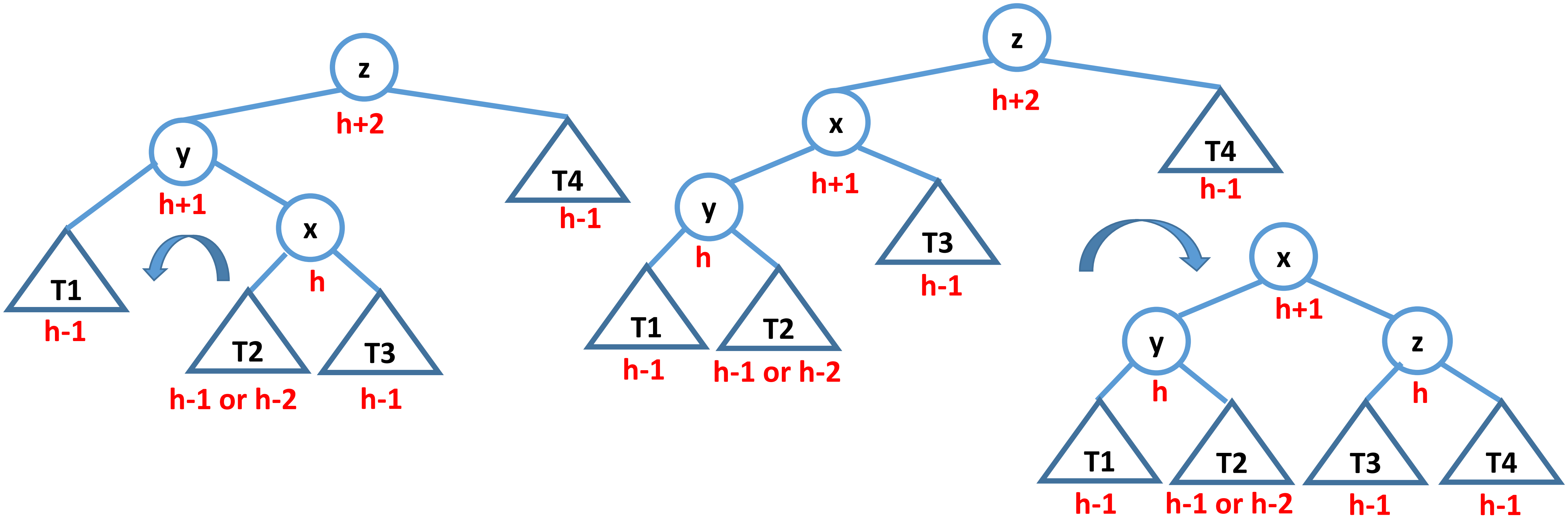
Double Rotation: Deletion at T4





Height of T2 or T3 must be h-1

Double Rotation: Deletion at T4



Height of final tree decreases ➡ Check for height imbalance upwards

Code for Deletion from an AVL Tree

```
node *deleteNode(node *root, int data)
{
  if(root == NULL)
        return NULL;
  else if(data < root->data)
        root->left = deleteNode(root->left, data);
  else if(data > root->data)
        root->right = deleteNode(root->right, data);
  else { //Code for deletion when the node has two children
  }
```

Code for Deletion: Node has Two Children

```
node *temp = NULL;
if(root->left != NULL && root->right != NULL) {
        temp = root->right;
        while(temp->left != NULL)     temp = temp->left;
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
        return root;
}
temp = (root->left != NULL) ? root->left : root->right;
free(root);
root = NULL;
return temp;
```

Left is Heavier than Right

```
root->height = findHeight(root);
int balanceFactorRoot = getBalanceFactor(root);
if(balanceFactorRoot == 2) {
        int balanceFactorLeft = getBalanceFactor(root->left);
        if(balanceFactorLeft == 0 || balanceFactorLeft == 1) {
                root = rightRotate(root);
        }
        else if(balanceFactorLeft == -1) {
                root->left = leftRotate(root->left);
                root = rightRotate(root);
        }
}
```

Right is Heavier than Left

```
else if(balanceFactorRoot == -2) {
        int balanceFactorRight = getBalanceFactor(root->right);
        if(balanceFactorRight == 0 || balanceFactorRight == -1) {
                root = leftRotate(root);
        }
        else if(balanceFactorRight == 1) {
                root->right = rightRotate(root->right);
                root = leftRotate(root);
        }
  }
  return root;
}
```

Analogy

- If you have 50 students, then at most 5 classrooms are needed.
- Min no of students that can be accommodated in 5 classrooms is 50
- Min no of nodes in an AVL tree of height h is c^h.
- If we have $n=c^h$ nodes, then the height is at most h
- If we have n nodes, then the height is at most $\log_c n$

Height of an AVL tree

- Different operations on a BST is O(h)
- In BST, h can be as bad as O(n)
- Need to design a better tree having height balanced property
- Result: AVL tree
- What is the height of an AVL tree in the worst case?
- AVL -> BST + Height balance property (Each node has |balance factor|<= 1)
- A -> B = Not(B) -> Not(A) Contrapositive statement

Height of an AVL Tree

Theorem: The height of an AVL tree storing n distinct keys is $O(\log_{1.414} n)$.

Proof:

- If there is at most n keys, then the height of the tree is ≤ h
- If the height of the tree is ≥ h, then there is at least n keys.

- To find the maximum height of an AVL tree with n keys, an easy way to approach the problem is, therefore, to find the minimum number of nodes n(h) in an AVL tree of height h.

- We see that n(1) = 2 and n(2) = 4.
- For h ≥ 3, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and the other AVL subtree of height h-2.
- We choose h-2 to consider minimum number of nodes in that subtree.

$$n(h) = 1 + n(h-1) + n(h-2)$$

Height of an AVL Tree

$$n(h) = 1 + n(h-1) + n(h-2)$$

$$n(h) > n(h-1) + n(h-2) > 2n(h - 2)$$

- We know that $n(h-1) > n(h-2)$. Therefore, we have
- $n(h) > 2n(h-2) > 2^2 n(h-4) > 2^3 n(h-6) > \ldots > 2^i n(h-2i)$
- When $i = h/2-1$, we have $n(h) > 2^{h/2-1} n(2) = 2^{h/2-1} . 4 = 2^{h/2+1} = 1.414^{h+2}$
- Taking logarithm on both sides, we have $\log_{1.414} n(h) > h+2 > h$
- $h < \log_{1.414} n(h)$.
- $n(h) < n$
- $h < \log_{1.414} n$
- Thus, the height of an AVL tree storing n distinct keys is $O(\log_{1.414} n)$.

Height of an AVL Tree: A Sharper Bound

Theorem: The height of an AVL tree storing n distinct keys is $O(\log_{1.618} n)$.

Proof:

- $n(h) = 1 + n(h-1) + n(h-2)$
- $n(1) = 2$ and $n(2) = 4$
- i.e. $1 + n(h) = 1+n(h-1) + 1+n(h-2)$
- Put $m(h) = 1+n(h)$
- i.e. $m(h) = m(h-1)+m(h-2)$ [$m(1) = 3$ $m(2) = 5$]
- Using the method of induction, we will prove that $m(h) \geq c^h$ for some $c>1$.
- **Basis Step**: If $h = 1$, then $n(1) = 2$ and $m(1) = 3 \geq c^1$ for some $c > 1$.
- If $h = 2$, then $n(2) = 4$ and $m(2) = 5 \geq c^2$ for some $c > 1$.

Height of an AVL Tree: A Sharper Bound

Induction Step:

- **Induction Hypothesis**: Suppose the claim is true for all $h < k$.
- We have to prove it for $h = k$
- i.e., we have to show that $m(k) \geq c^k$ for some $c>1$.
- i.e. we have to show that $m(k-1) + m(k-2) \geq c^k$
- i.e., we have to show that $c^{k-1} + c^{k-2} \geq c^k$, [From Induction Hypothesis]
- i.e., we have to show that $c^2-c-1 \leq 0$,
- i.e., $c \leq (1+\sqrt{5})/2 = 1.618$ (golden ratio).
- The solution of $m(h) = m(h-1) + m(h-2)$ is 1.618^h
- Minimum no of nodes that can be accommodated in an AVL tree of height h is 1.618^h .
- Hence, the height of an AVL tree storing n distinct keys is at most $\log_{1.618} n$.