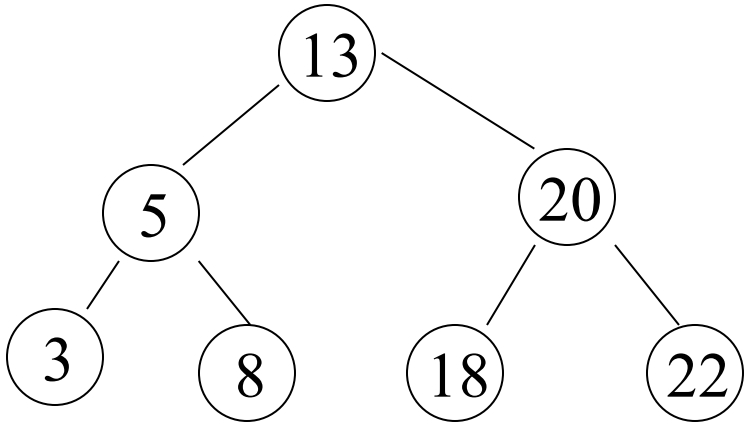
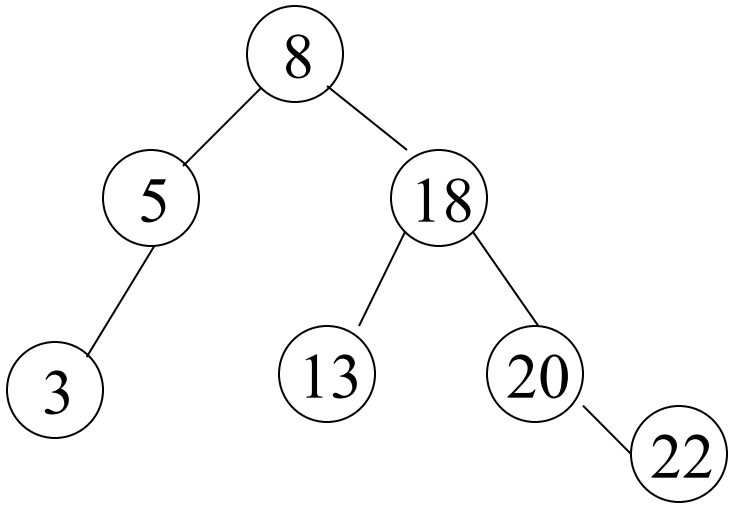
AVL Tree

IIITS

Motivation

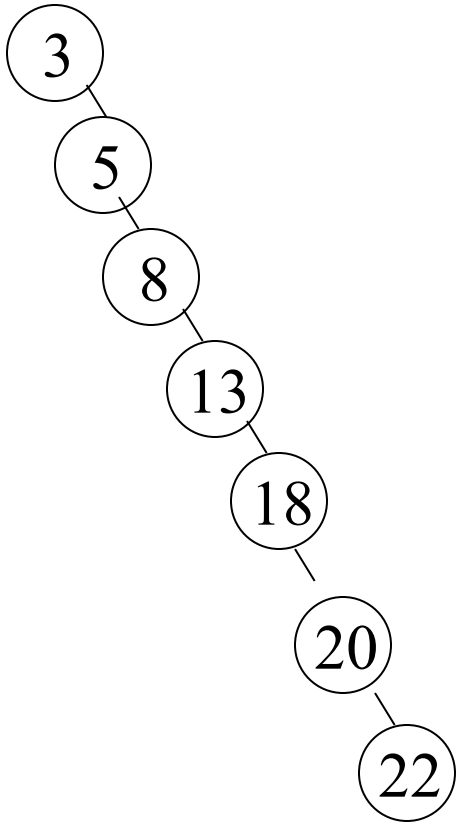
- Complete binary tree is hard to build when we allow dynamic insert and remove.
  - We want a tree that has the following properties
    - Tree height = O(log(N))
    - allows dynamic insert and remove with O(log(N)) time complexity.
  - The AVL tree is one of this kind of trees.

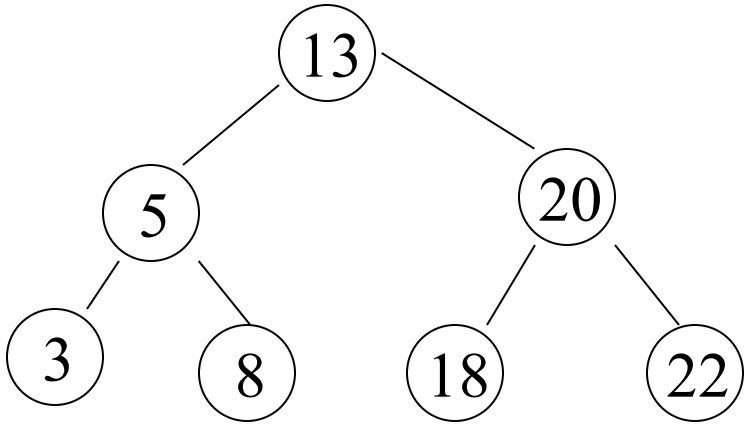




Motivation

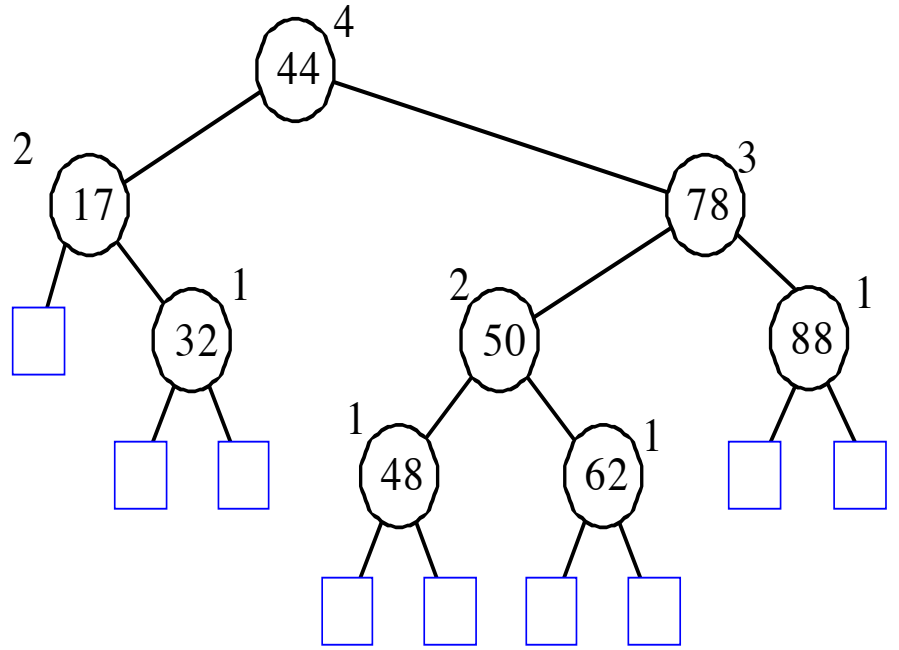
- When building a binary search tree, what type of trees would we like? Example: 3, 5, 8, 20, 18, 13, 22





AVL (Adelson-Velskii and Landis) Trees

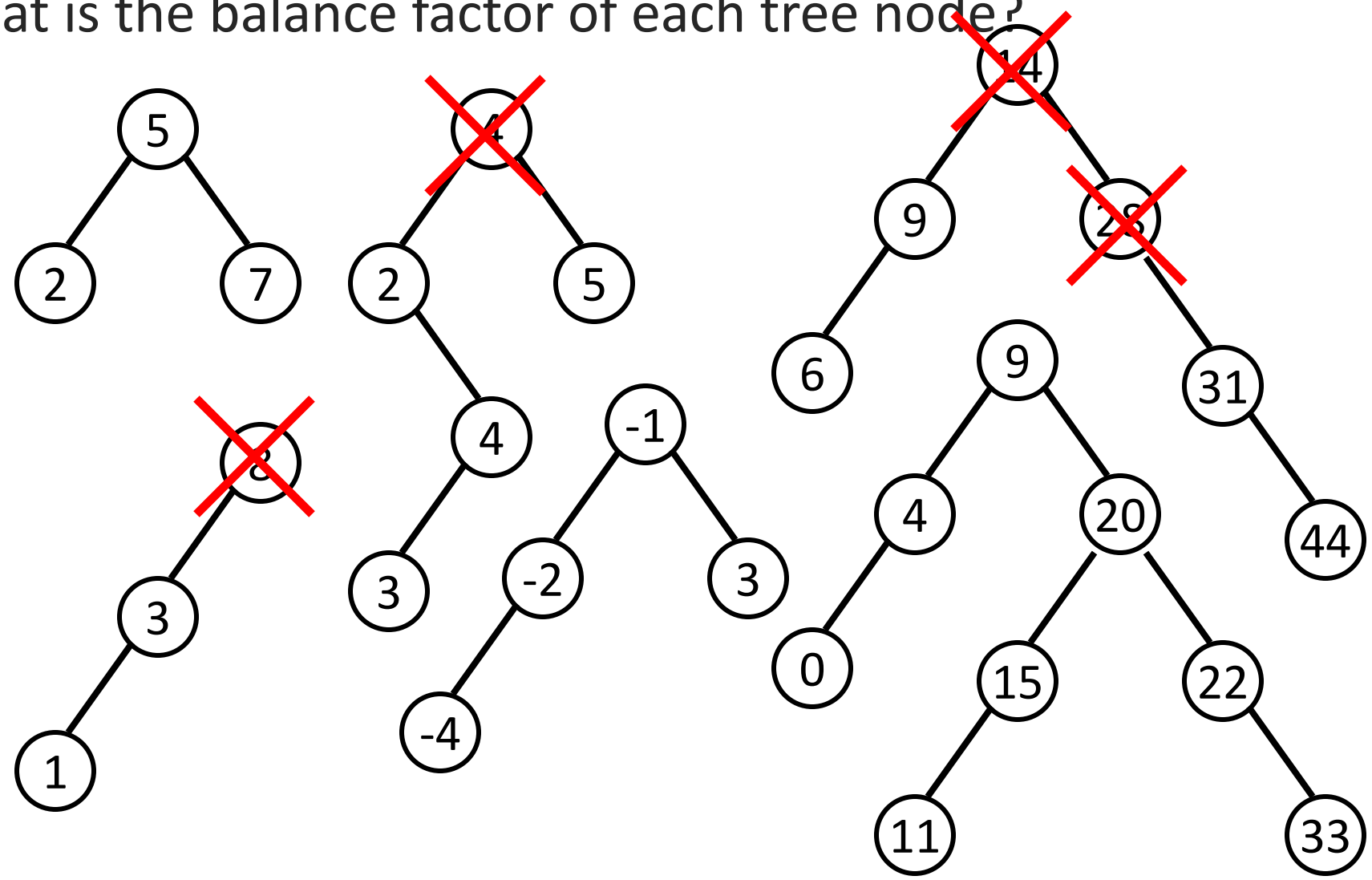
An AVL Tree is a self-balancing ***binary search tree*** such that for every internal node *v* of *T*, the *heights of the children of v can differ by at most 1.*



An example of an AVL tree where the heights are shown next to the nodes:

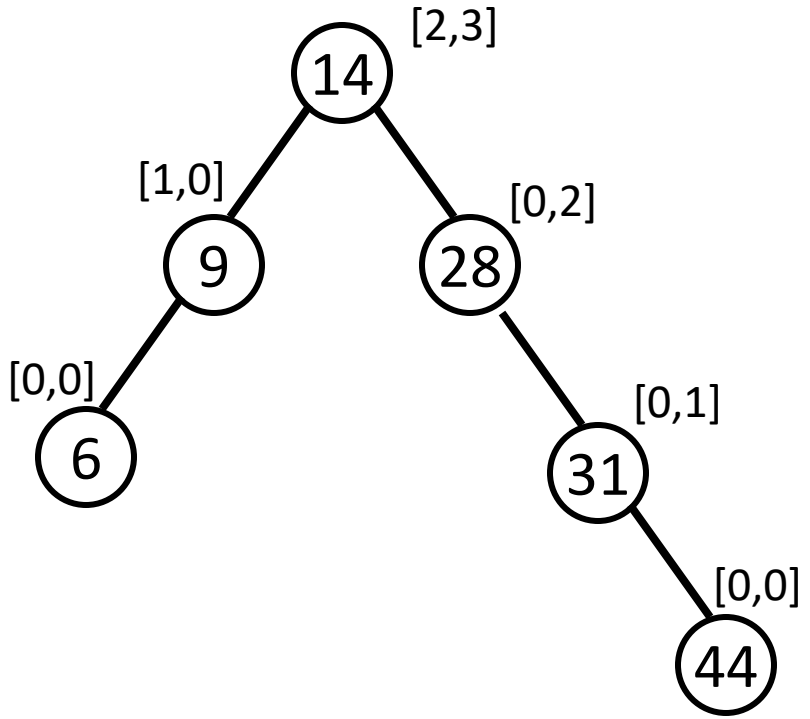
Which are valid AVL trees?

- What is the balance factor of each tree node?



Which are valid AVL trees?

- What is the balance factor of each tree node?

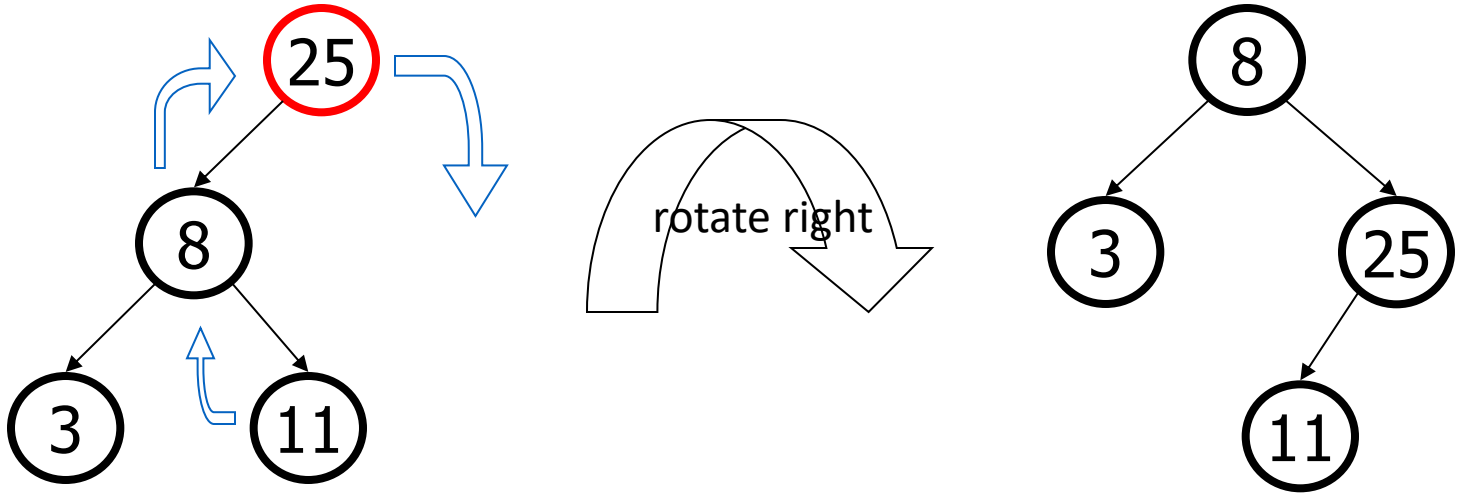


AVL Tree Insert and Remove

- Do binary search tree insert and remove
- The balance condition can be violated sometimes
  - Do something to fix it – rebalancing – rotations the magic!
  - After rotations, the balance of the whole tree is maintained

Key idea: rotations

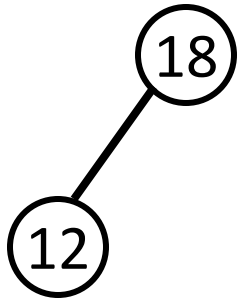
- If a node has become out of balanced in a given direction, *rotate* it in the opposite direction.
  - *rotation:* A swap between parent and left or right child, maintaining proper BST ordering.



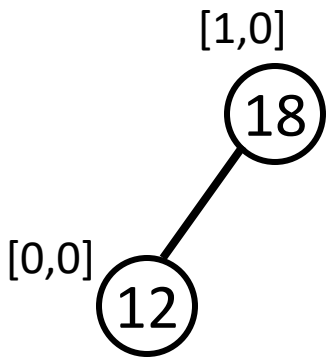
Balance Condition Violation

- If condition violated after a node insertion or deletion if the tree is not balanced then following 4 rotation cases might occur.
  1. Left - Left
  2. Right - Right
  3. Left - Right
  4. Right - Left

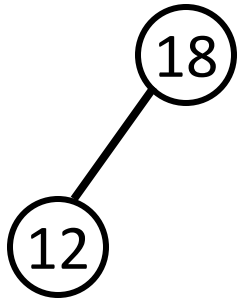
Left-Left Case



Left-Left Case

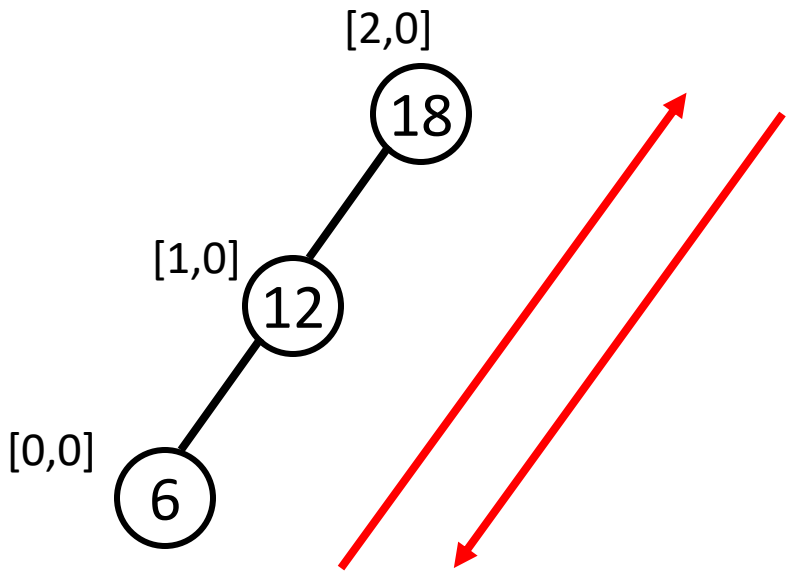


Left-Left Case



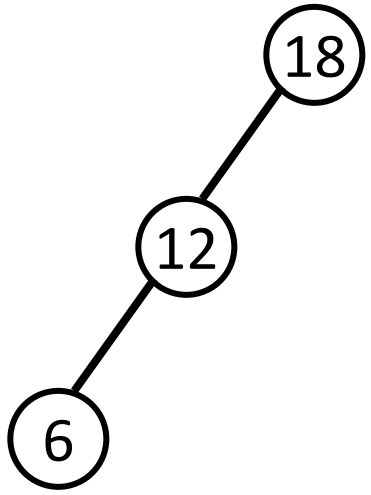
Insert 6

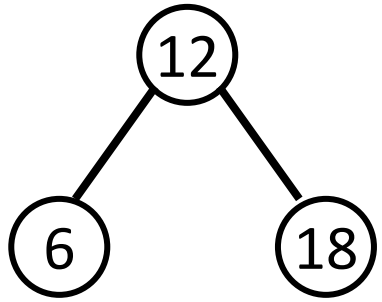
Left-Left Case



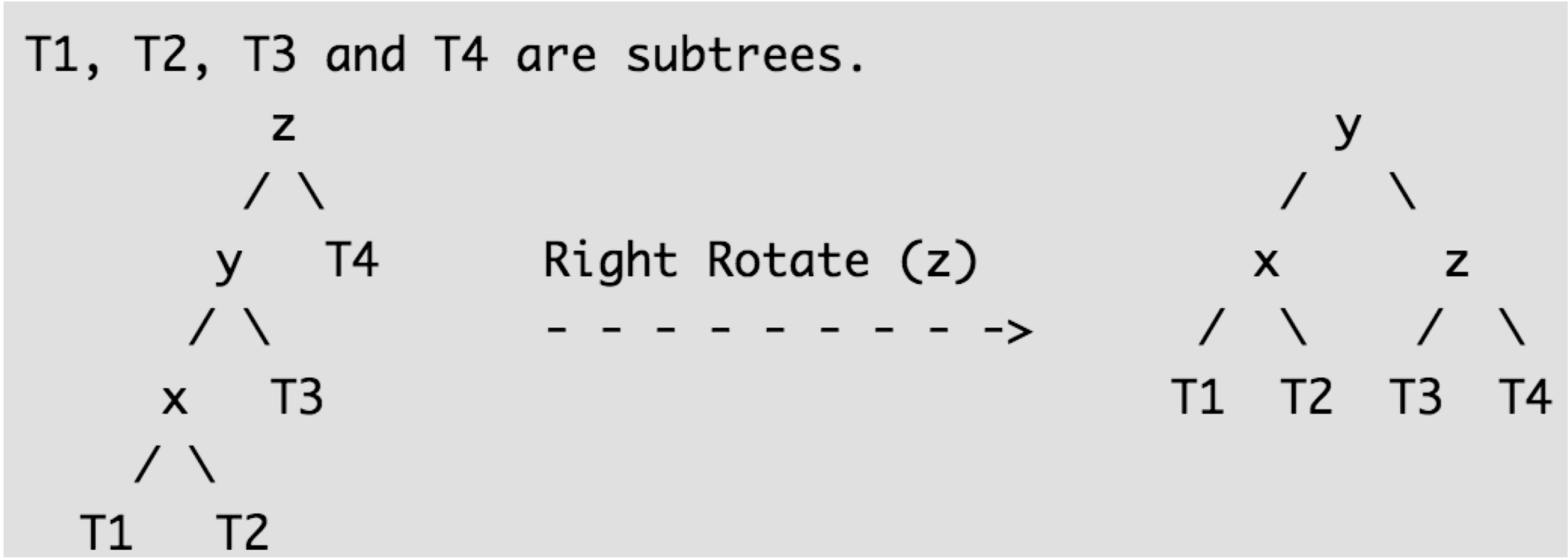
Count first two move – Left Left

Left-Left Case – Right Rotation





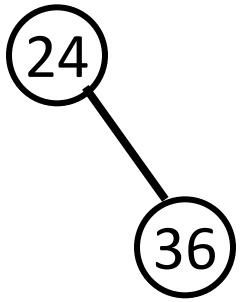
Left-Left Complex Case



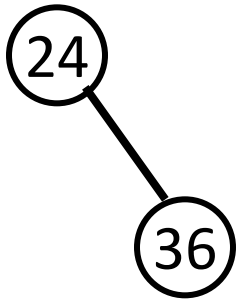
Left-Left Complex Case – Right Rotation

```
// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Return new root
    return x;
}
```

Right-Right Case

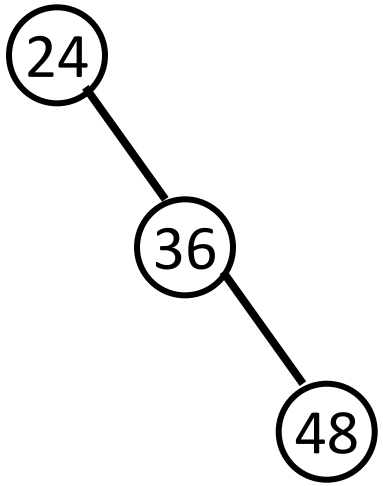


Right-Right Case

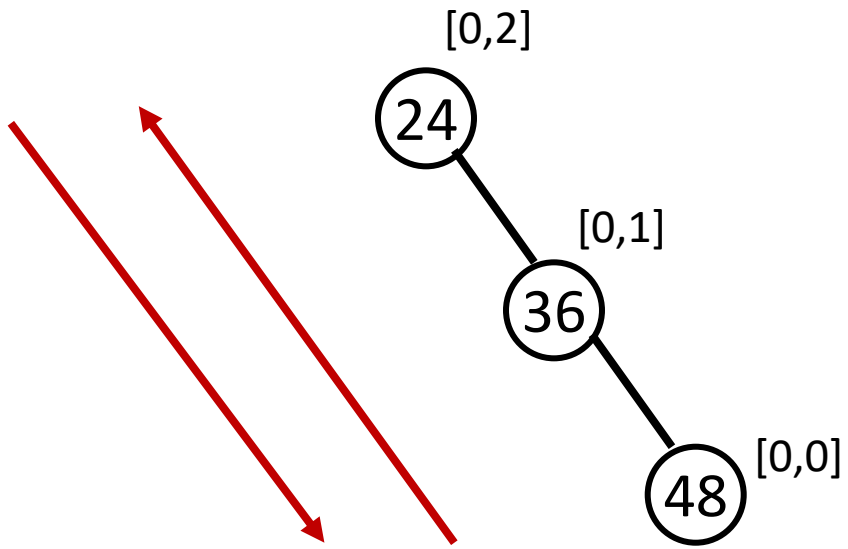


Insert 48

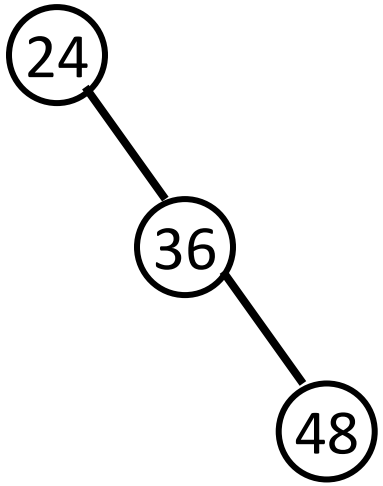
Right-Right Case

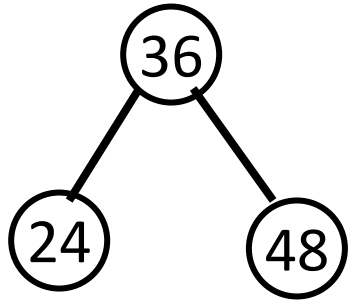


Right-Right Case

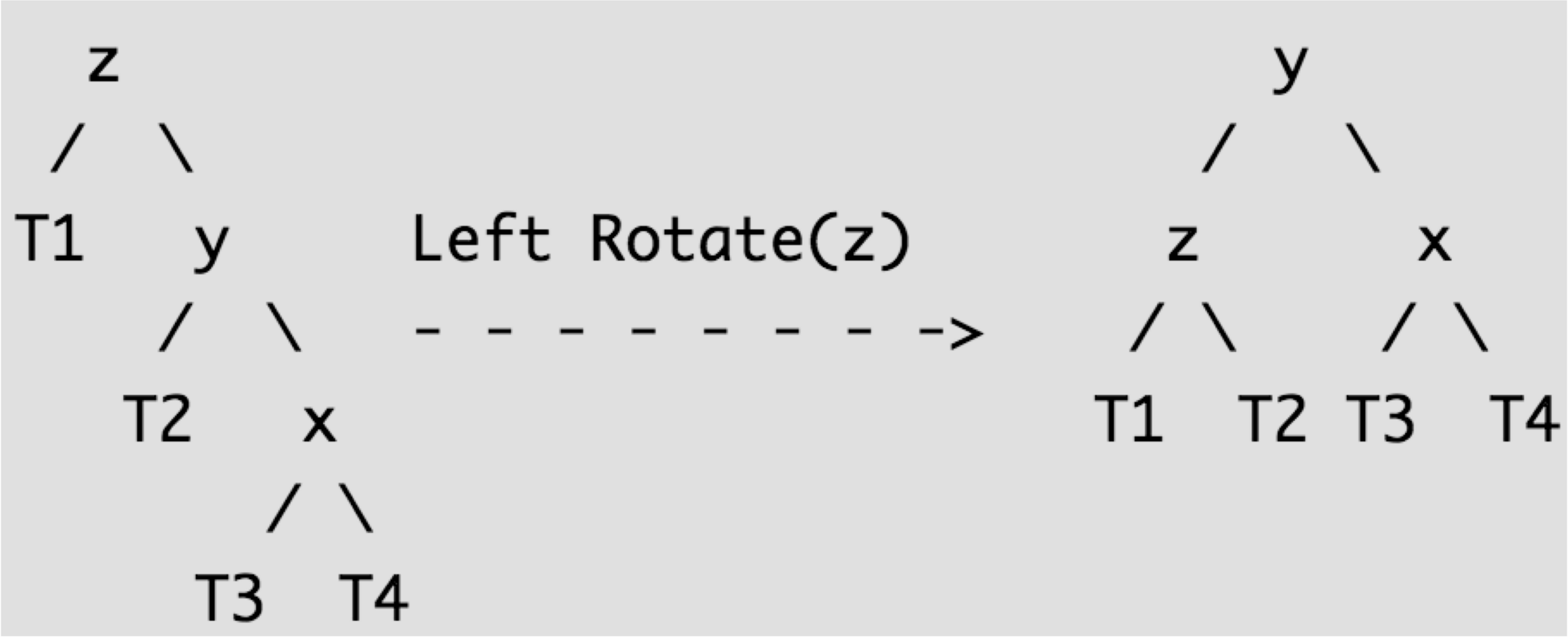


Right-Right Case – Left Rotation





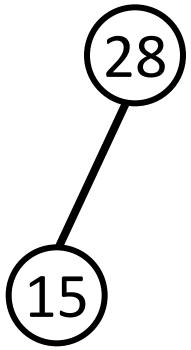
Right-Right Case – Complex Case



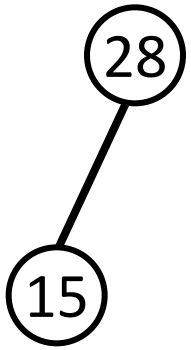
Right-Right Case – Left Rotation

```
// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    //  Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}
```

Left-Right Case

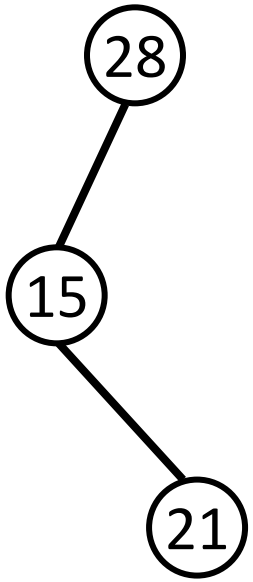


Left-Right Case



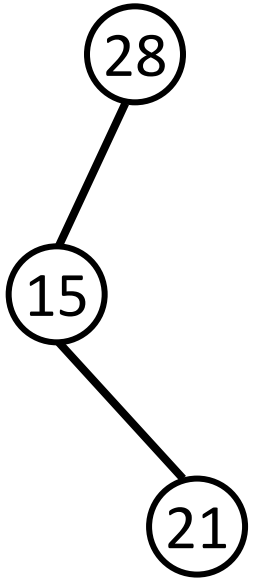
Insert 21

Left-Right Case



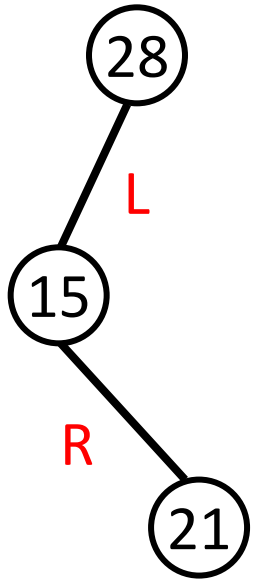
Left-Right Case

- Two problems
- Two rotations



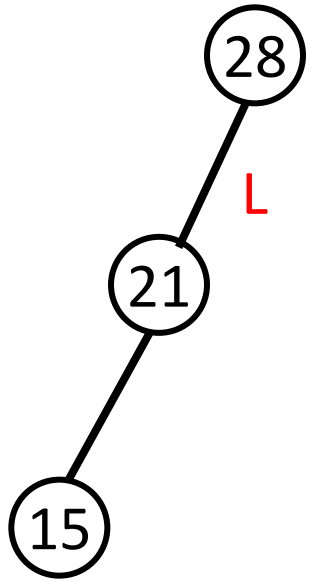
Left-Right Case

- Two problems
- Two rotations



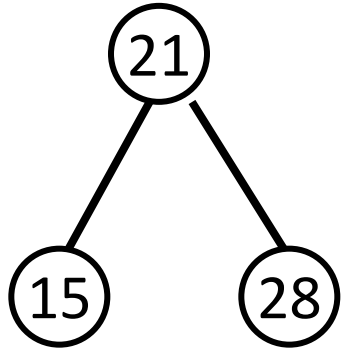
Left-Right Case – Right Rotation

- Two problems
- Two rotations



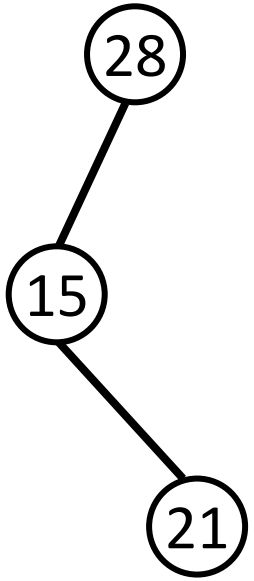
Left-Right Case – Right-Left Rotation

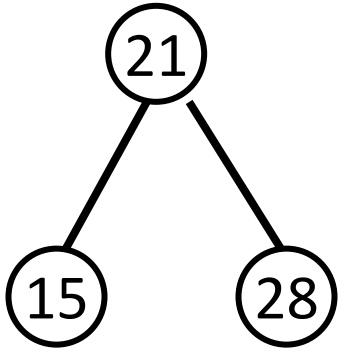
- Two problems
- Two rotations



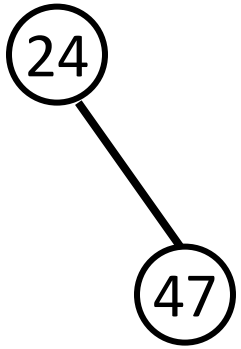
Left-Right Case

- Two problems
- Two rotations

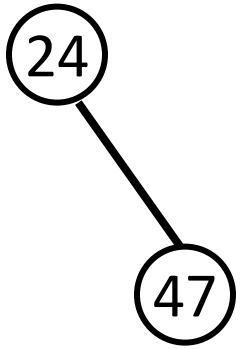




Right-Left Case

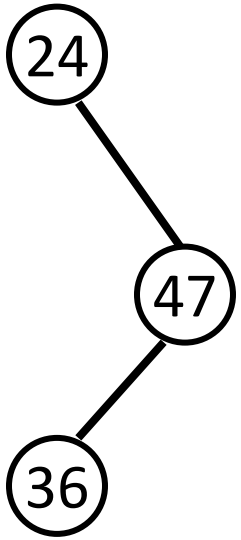


Right-Left Case



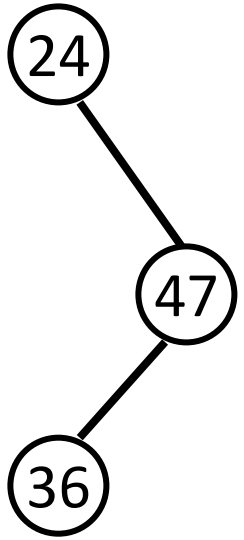
Insert 36

Right-Left Case



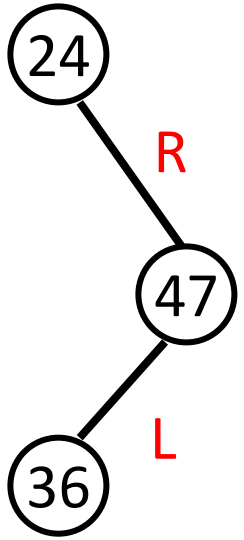
Right-Left Case

- Two problems
- Two rotations



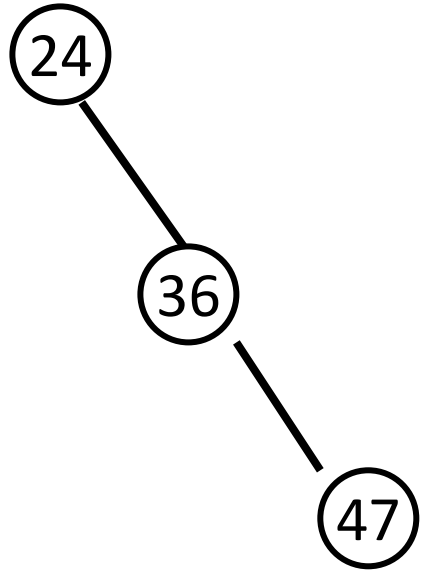
Right-Left Case

- Two problems
- Two rotations



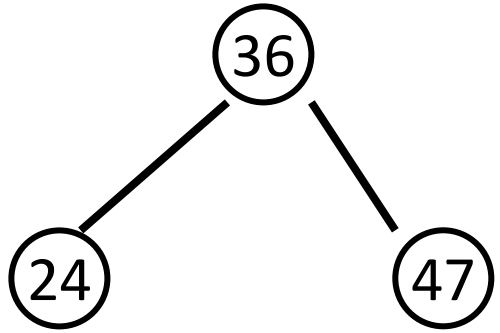
Right-Left Case – Right Rotation

- Two problems
- Two rotations



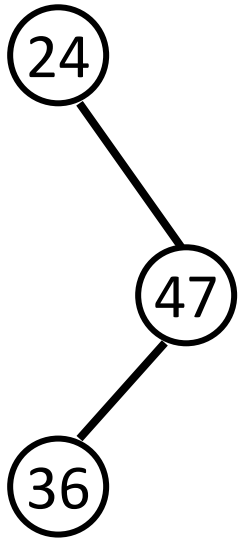
Right-Left Case – Left-Right Rotation

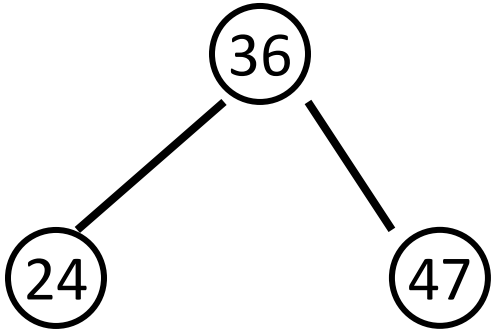
- Two problems
- Two rotations



Right-Left Case – Left-Right Rotation

- Two problems
- Two rotations





See a complete example

21 26 30 9 4 14 28 18 15 10 2 3 7

AVL Tree Insertion

```
// Recursive function to insert key in subtree rooted
// with node and returns new root of subtree.
struct Node* insert(struct Node* node, int key)
{
    /* 1.  Perform the normal BST insertion */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;
```

AVL Tree Insertion

```
// Recursive function to insert key in subtree rooted
// with node and returns new root of subtree.
struct Node* insert(struct Node* node, int key)
{
    /* 1.  Perform the normal BST insertion */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;
```

AVL Tree Insertion

```
// Get Balance factor of node N
int getBalance(struct Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}
```

AVL Tree Insertion

```
// If this node becomes unbalanced, then
// there are 4 cases

// Left Left Case
if (balance > 1 && key < node->left->key)
    return rightRotate(node);

// Right Right Case
if (balance < -1 && key > node->right->key)
    return leftRotate(node);

// Left Right Case
if (balance > 1 && key > node->left->key)
{
    node->left =  leftRotate(node->left);
    return rightRotate(node);
}

// Right Left Case
if (balance < -1 && key < node->right->key)
{
    node->right = rightRotate(node->right);
    return leftRotate(node);
}
```

AVL Tree Deletion

- Deletion rules of BST
- Check for balance – re-balance if necessary

AVL Tree Insertion - Reference

- https://www.geeksforgeeks.org/avl-tree-set-1-insertion/
- https://www.geeksforgeeks.org/avl-tree-set-2-deletion/

AVL Tree - Complexity

- An AVL tree is balanced, so its height is O(log N) where N is the number of nodes.
- The rotation routines are all themselves O(1), so they don't significantly impact the insert operation complexity, which is still O(k) where k is the height of the tree. But as noted before, this height is O(log N), so insertion into an AVL tree has a worst case O(log N).
- Searching an AVL tree is completely unchanged from BST's, and so also takes time proportional to the height of the tree, making O(log N).
- Removing nodes from a binary tree also requires rotations, but remains O(log N) as well.