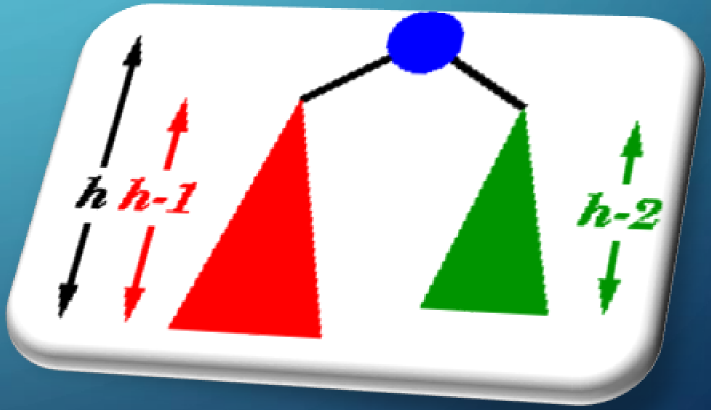
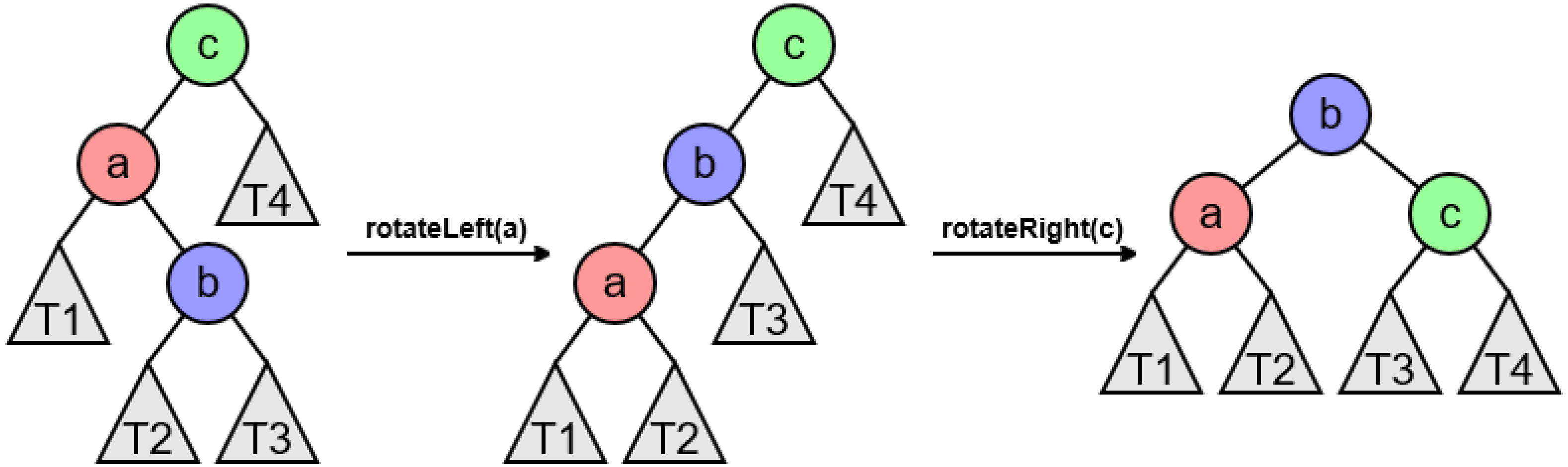
AVL TREE BALANCING

ADELSON-VELSKII AND LANDIS



Adelson-Velskii and Landis Trees

An *AVL tree* is a self-balancing binary search *tree*. In an *AVL tree*, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.



Why AVL Tree?

- Most of the BinarySearchTree operations (i.e., search, max, min, insert, delete.. etc) take O(H) time where H is the height of the BST.
- The cost of these operations may become O(N) for a skewed Binary tree.
- If the height of the tree remains O(LogN) after every insertion and deletion, then the upper bound of O(LogN) for all these operations.
- The height of an AVL tree is always O(LogN) where n is the number of nodes in the tree.

Rebalancing Algorithm

Rebalance the tree by performing appropriate rotations on the subtree rooted with z. There are 4 cases that need to be handled as x, y and z:

- y is left child of z and x is left child of y (Left-Left Case)
- y is left child of z and x is right child of y (Left-Right Case)
- y is right child of z and x is right child of y (Right-Right Case)
- y is right child of z and x is left child of y (Right-Left Case)

Left-Left

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1   T2 T3   T4
    / \
  T1   T2
```

Left-Right

```
     z                                z                              x
    / \                             /   \                          /  \
   y   T4  Left Rotate (y)         x     T4  Right Rotate(z)     y      z
  / \      - - - - - - - - ->     /  \       - - - - - - - ->   / \    / \
T1   x                           y    T3                      T1  T2 T3  T4
    / \                         / \
  T2   T3                     T1   T2
```

Right-Right

```
  z                                y
 /  \                            /   \ 
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - - ->  / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

Right-Left

```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /  \   - - - - - - - ->  / \    / \
   x   T4                      T2   y                  T1  T2  T3  T4
  / \                              /  \
T2   T3                           T3   T4
```

Terminology

- The ***root node*** is the first node above the inserted node that is unbalanced.

- The ***child node*** is either the left or right child of the root node. The child node involved in rotation is the node on the path towards the recently inserted value.

Rotating Left

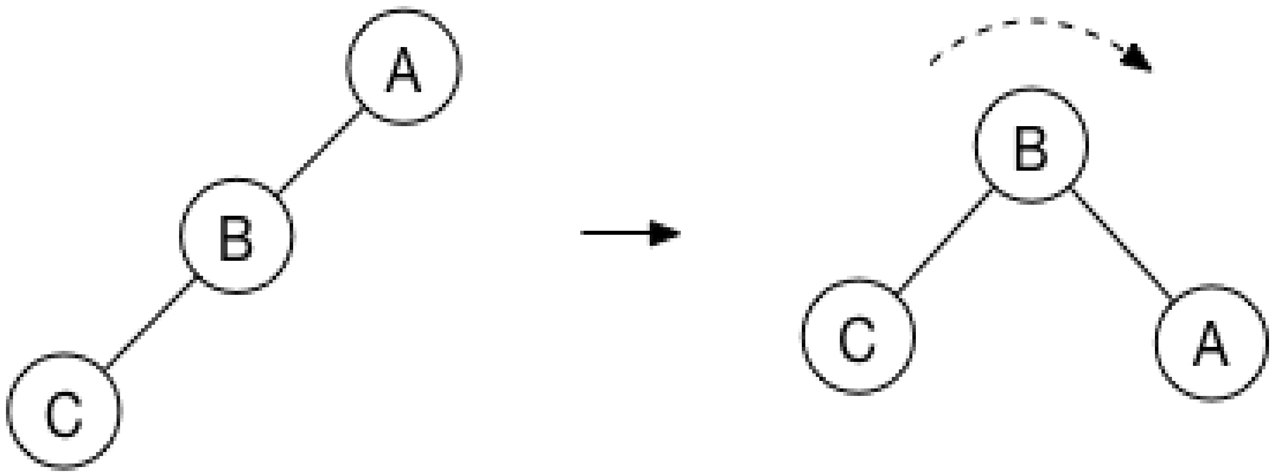
1. Save value of pivot.right (temp = pivot. right)
2. Set root.right to value of pivot.right.left
3. Set temp.left to pivot
4. Set pivot to temp

Rotating Right

1. Save value of pivot.left (temp = pivot.left)
2. Set pivot.left to value of pivot.left.right
3. Set temp.right to pivot
4. Set pivot to temp

LEFT-LEFT

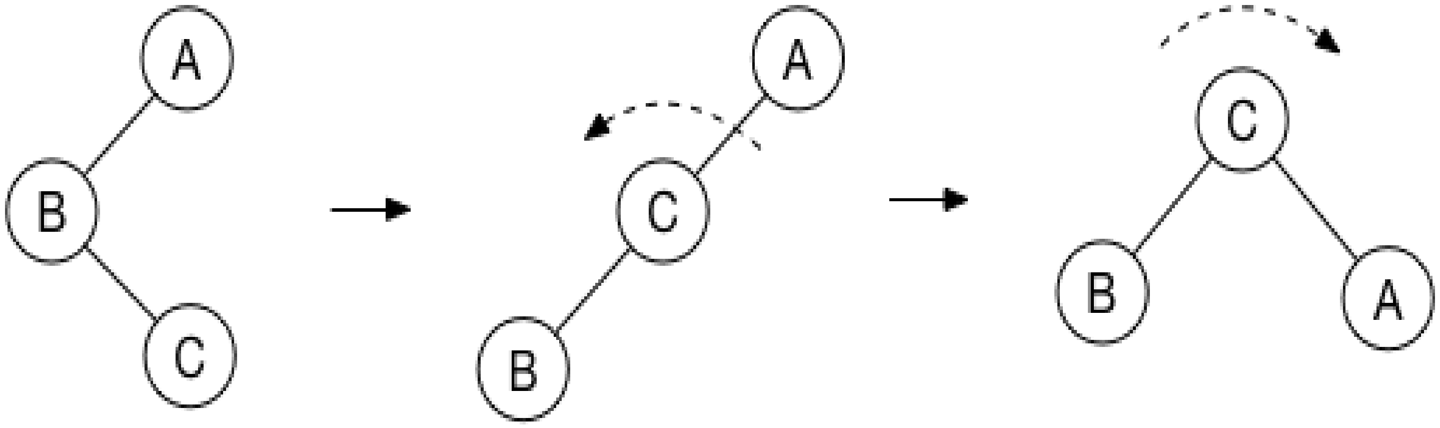
- *The left sub-tree of the left child grew.*
- To balance: Right rotation around the root.



```
// Left- Left Rotation
Node<T> ll_rotation(Node<T>* parent)
        Node<T> temp = parent.left
        parent.left = temp.right
        temp.right = parent
        return temp
```

LEFT-RIGHT

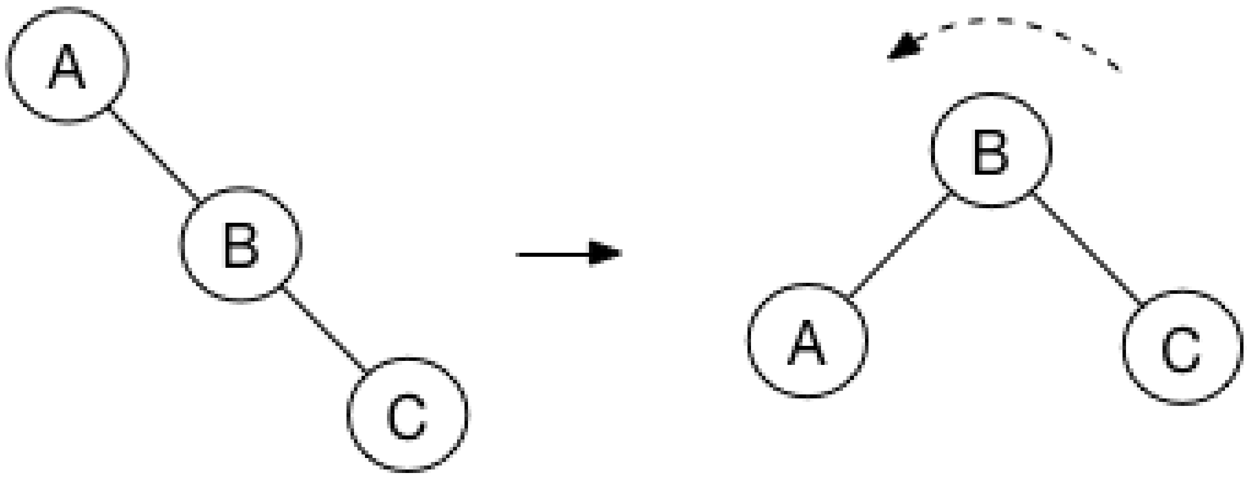
- *The right sub-tree of the left child grew.*
- To balance: Left rotation around child, *then r*ight rotation around root.



```
// Left - Right Rotation
Node<T> lr_rotation(Node<T> parent)
        Node<T> tempL = parent.left
        parent.left = rr_rotation(tempL)
        return ll_rotation(parent)
```

RIGHT-RIGHT

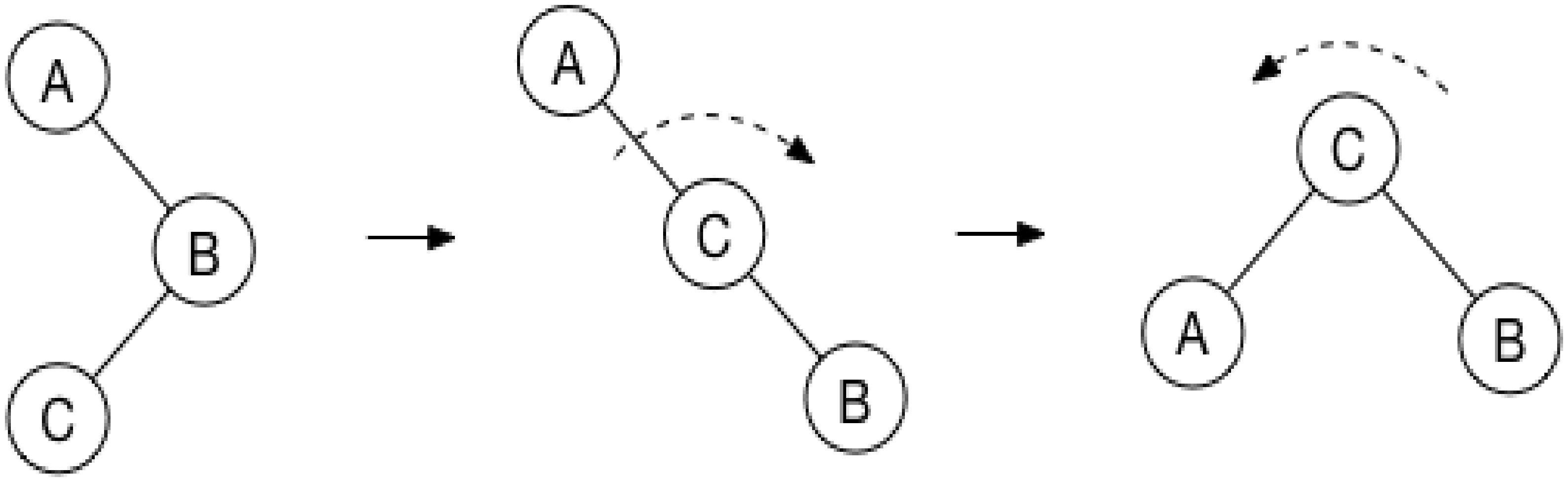
- *The right sub-tree of the right child grew.*
- To balance: Left rotation around root.



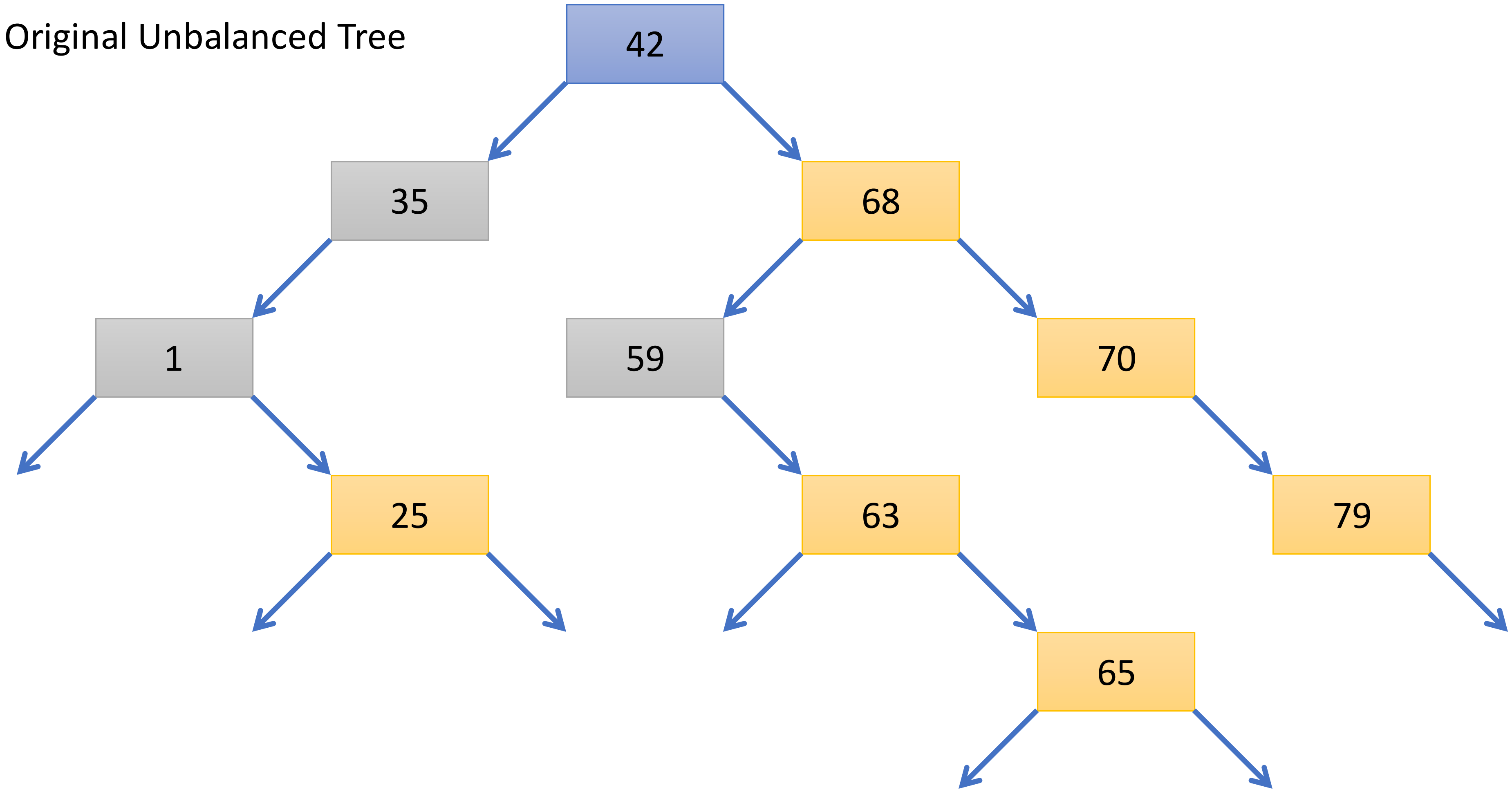
```
// Right- Right Rotation
Node<T> rr_rotation(Node<T> parent)
        Node<T> temp = parent.right
        parent.right = temp.left
        temp.left = parent
        return temp
```

RIGHT-LEFT

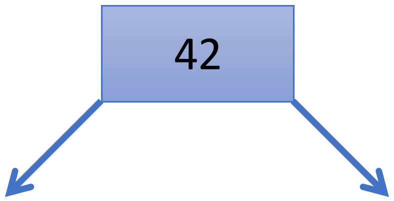
- *The left sub-tree of the right child grew.*
- To balance: right rotation around child, *then* left rotation around root.



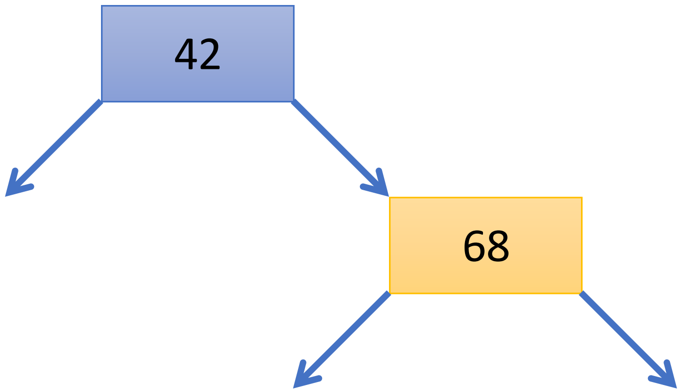
```
// Right- Left Rotation
Node<T> rl_rotation(Node<T> parent)
        Node<T> temp = parent.right
        parent.right = ll_rotation(temp)
        return rr_rotation(parent)
```


Original Unbalanced Tree
42
35
68
1
59
70
25
63
79
65

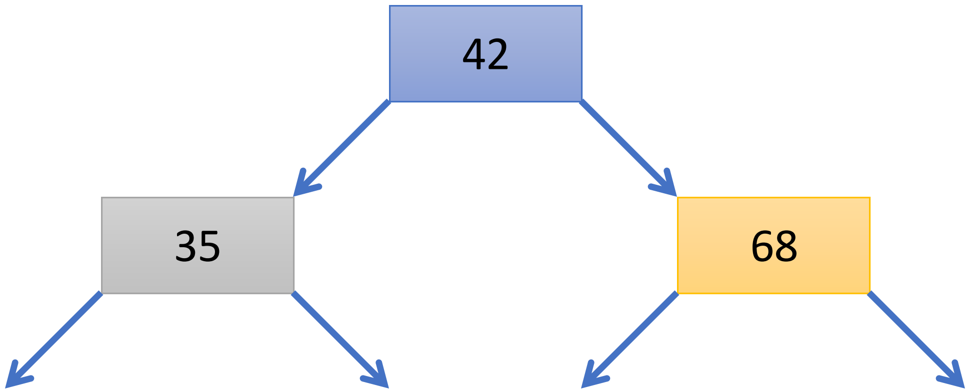
Insert 42



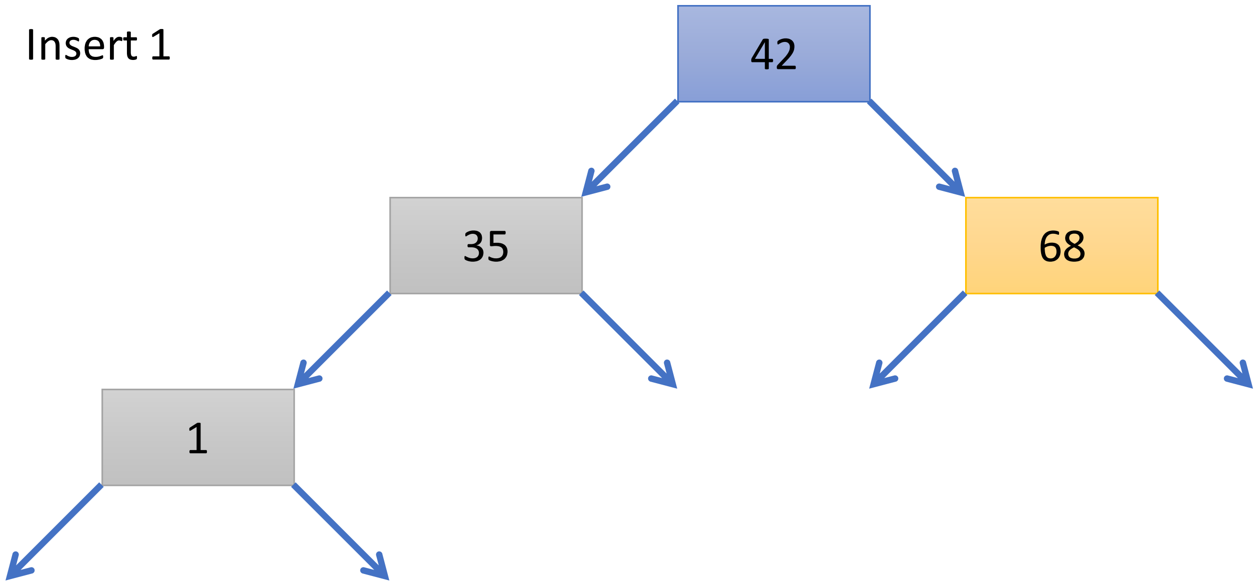
Insert 68



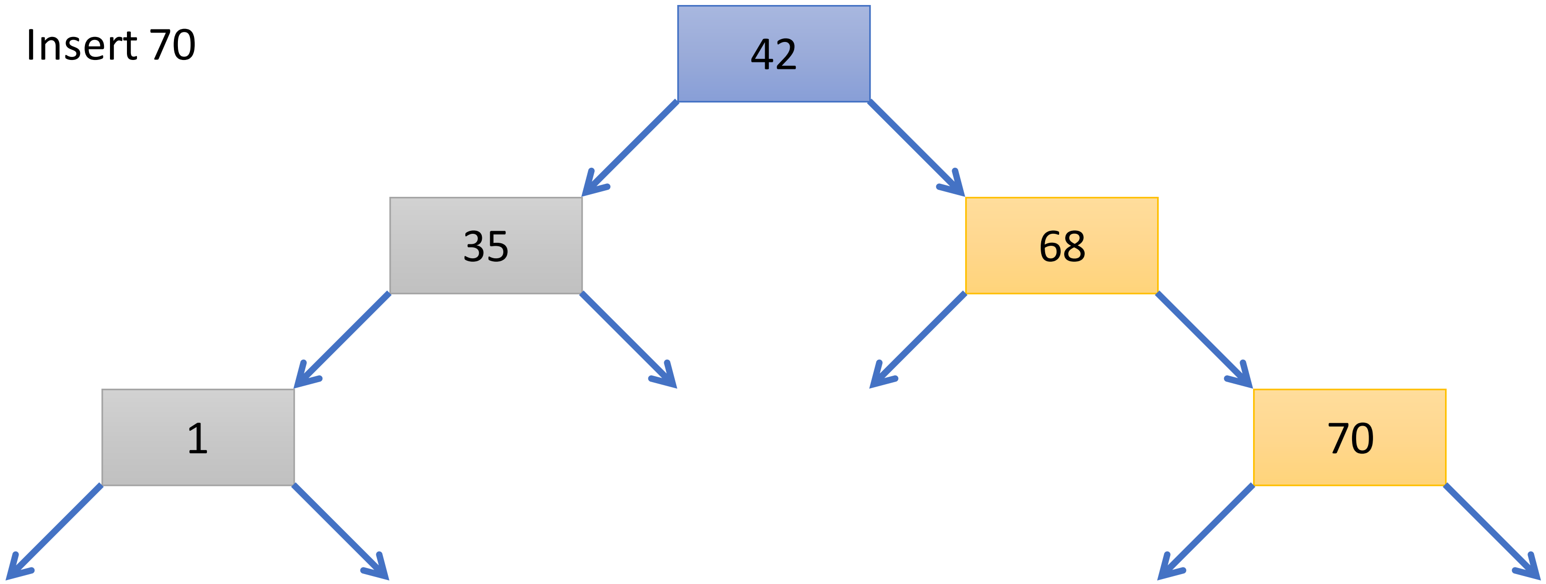
Insert 35



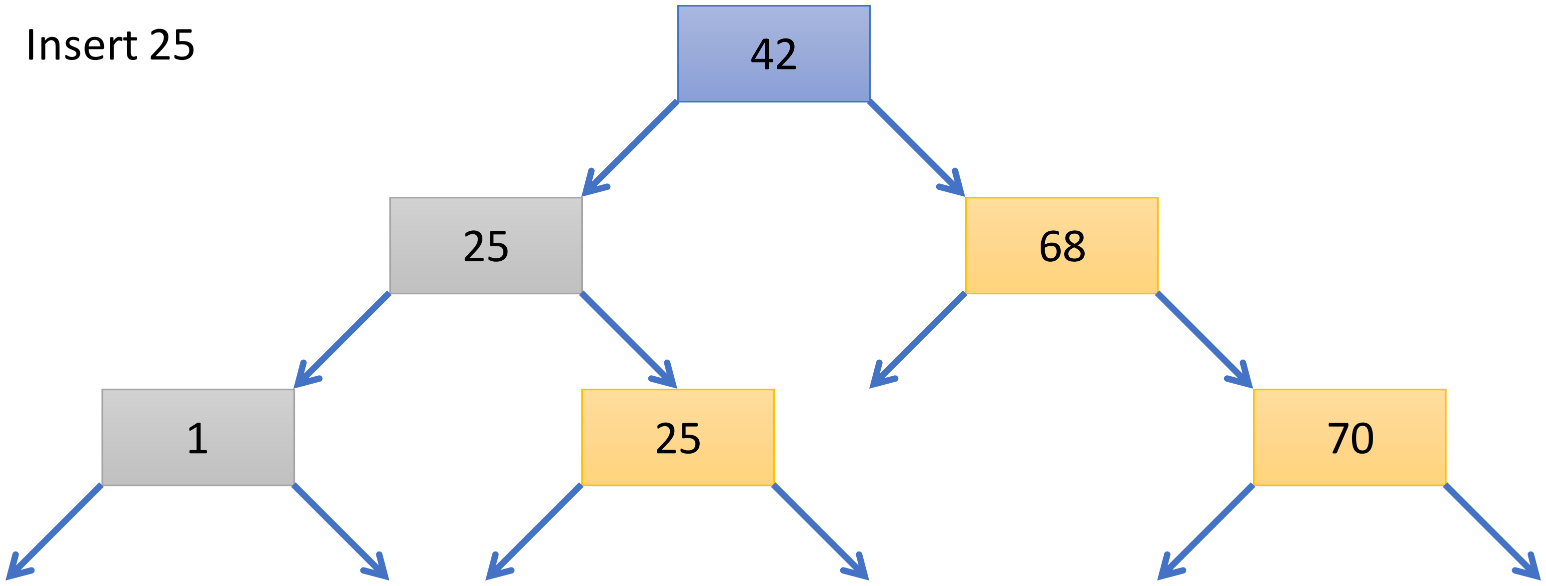
Insert 1



Insert 70



Insert 25



lr_rotation parent data: 35

  parent->left: 1

rr_rotation parent data: 1

  parent->right data: 25

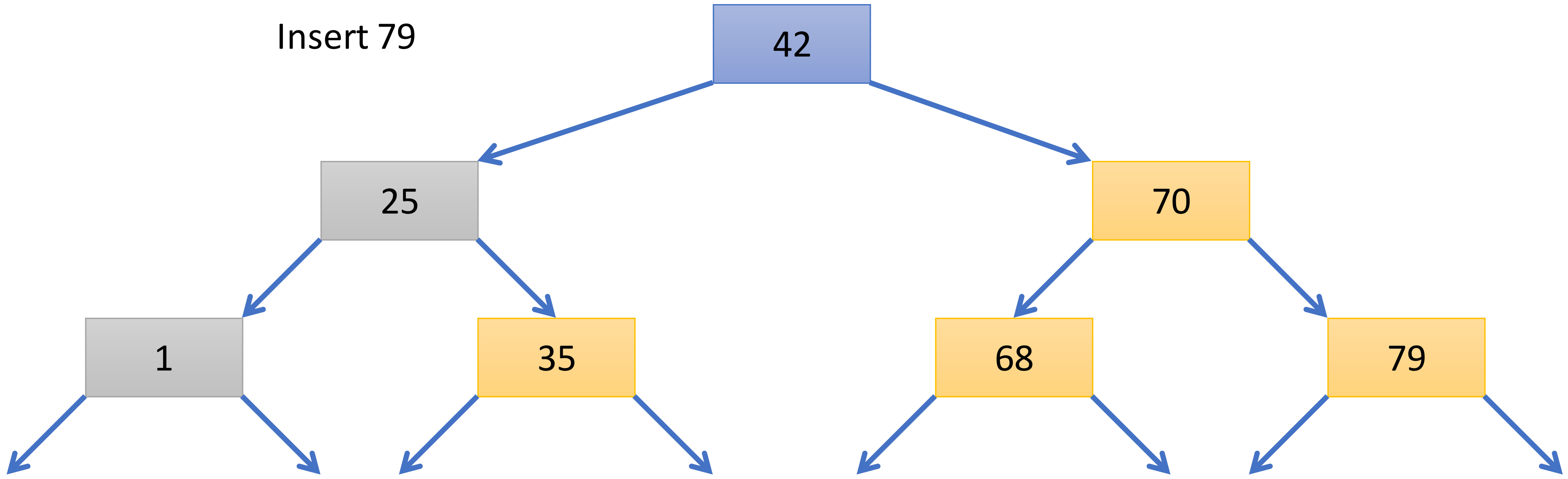
  parent->left data: 1

  parent->left after rr_rotation: 25

ll_rotation: 35

  parent->left: 25

Insert 79

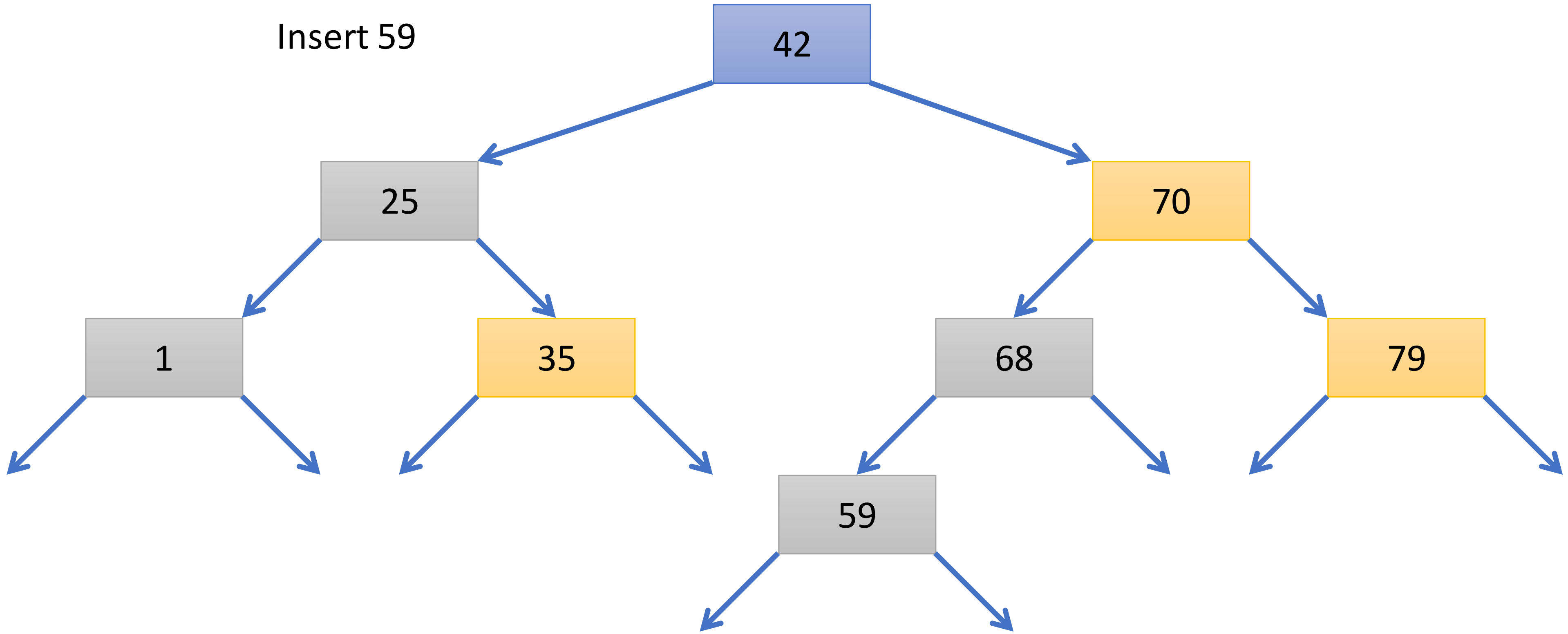


rr_rotation parent data: 68

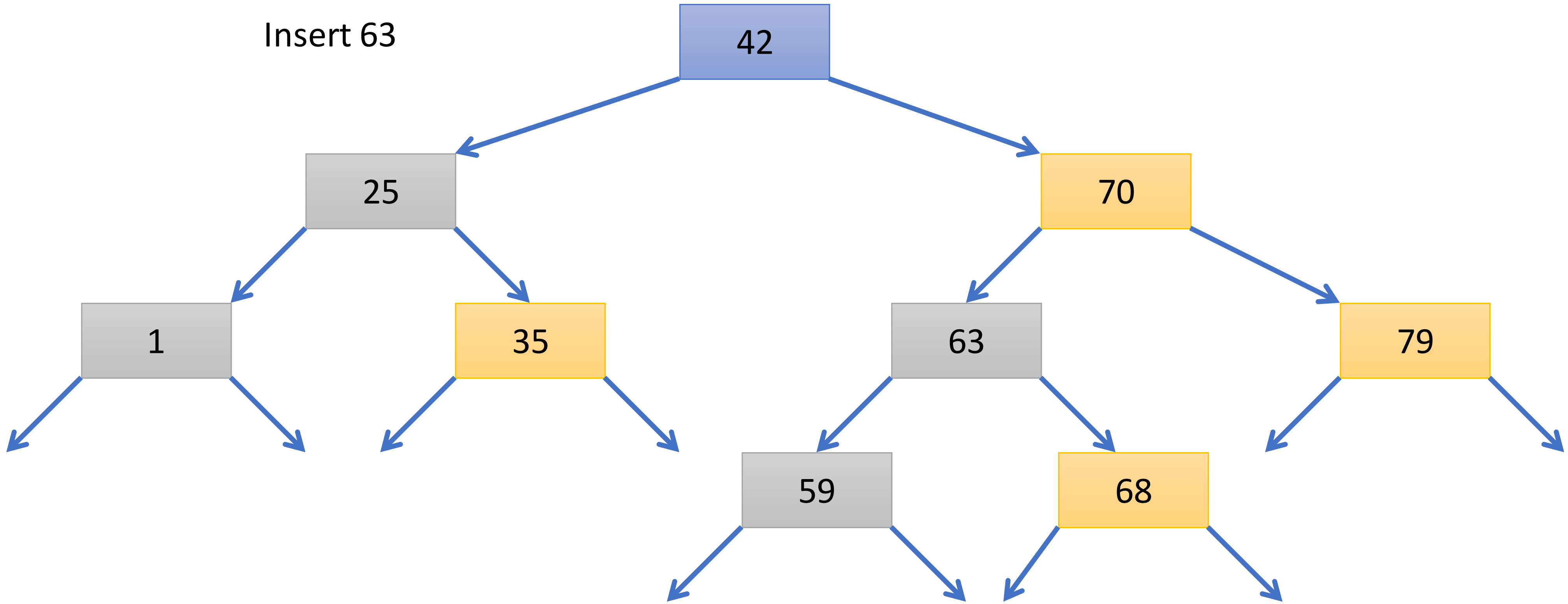
parent->right data: 70

parent->left data: 68

Insert 59

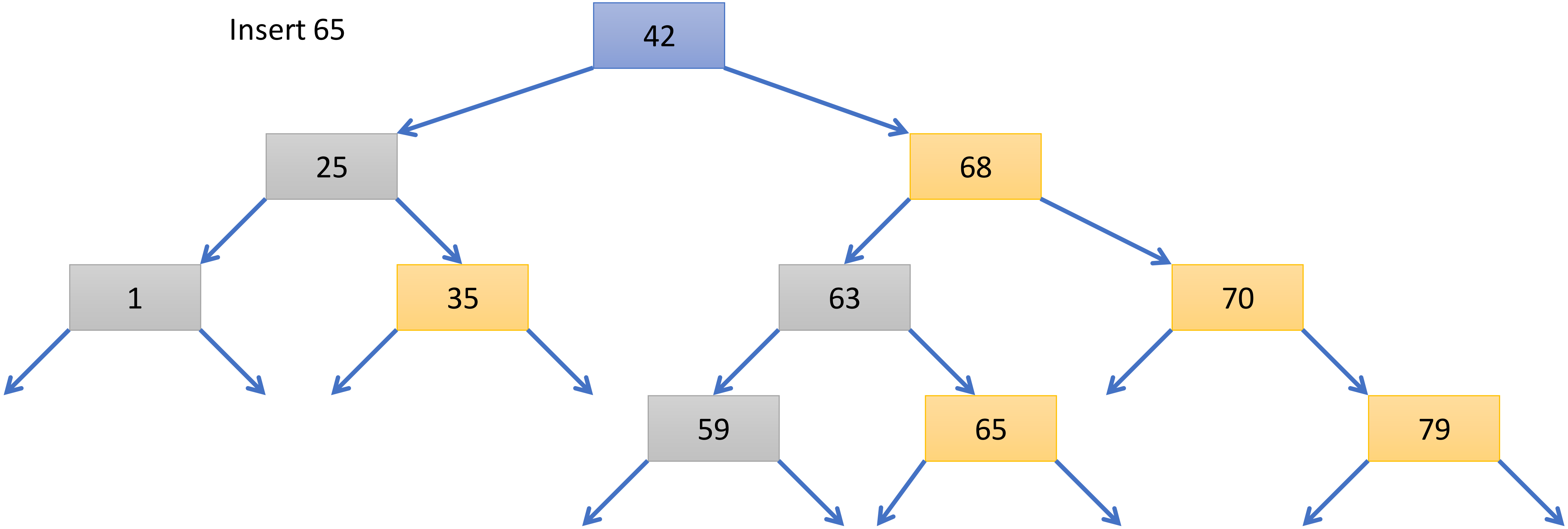


Insert 63



```
lr_rotation parent data: 68
  parent->left: 59
rr_rotation parent data: 59
  parent->right data: 63
  parent->left data: 59
  parent->left after rr_rotation: 63
ll_rotation: 68
  parent->left: 63
```

Insert 65



lr_rotation parent data: 70

parent->left: 63

rr_rotation parent data: 63

parent->right data: 68

parent->right data: 65

parent->left data: 63

parent->left after rr_rotation: 68

ll_rotation: 70

parent->left: 68

```
node insert<T>(T e, node<T> n )
    if( e < n.data )
        n.left = insert( e, n.left )
        if( height( n.left ) - height( n.right ) == 2 )
            if( e < n.left.data )
                n = ll_rotation( n )
            else
                n = rl_rotation( n )
    else if( e > n.data )
        n.right = insert( e, n.right )
        if( height( n.right ) - height( n.left ) == 2 )
            if( e > n.right.data )
                n = rr_rotation( n )
            else
                n = lr_rotation( n )
    n.height = max( height( n.left ), height( n.right ) ) + 1
    return n
```

Time Complexity

The rotation operations (left and right rotate) take constant time as only few pointers are being changed there:

- Updating the height and getting the balance factor take constant time. The time complexity of AVL insert remains same as BST insert which is O(H) where H is height of the tree.

- Since AVL tree is balanced, the height is O(LogN). The time complexity of AVL insert is O(LogN).