Balanced Trees

Semester 2, 2020

Kris Ehinger

Binary search trees

- Good average case behaviour: O(log n)
- Bad worst case behaviour: O(n)
- Recall that we usually base time complexity on worst case, so overall: O(n)

Binary search trees

- Solution? How to get a BST to stay balanced?
  - Or almost balanced?
  - Regardless of the data order?

- Balanced trees: AVL, red-black; 2,3,4; B+tree

Balanced trees

- Method to ensure BST is perfectly balanced (or almost balanced)
- Why? Keeps the height of the tree O(log n)
  - Perfectly balanced tree, height = log n, exactly
  - Approximately balanced tree, height = O(log n)
- Importantly, method should not increase time complexity to build tree
  - Search one item in a balanced tree: O(log n)
  - Build a balanced tree of n items: O(n log n)

Balanced trees

- Add steps during insertion to ensure the tree does not become unbalanced
  - Binary search tree ordering is preserved: left child < parent, right child > parent
- So, search in a balanced tree is exactly the same as binary tree
  - But search time is guaranteed to be O(log n)

Balanced trees

- AVL trees
- 2-3-4 trees
- B+ trees
- Red-black trees

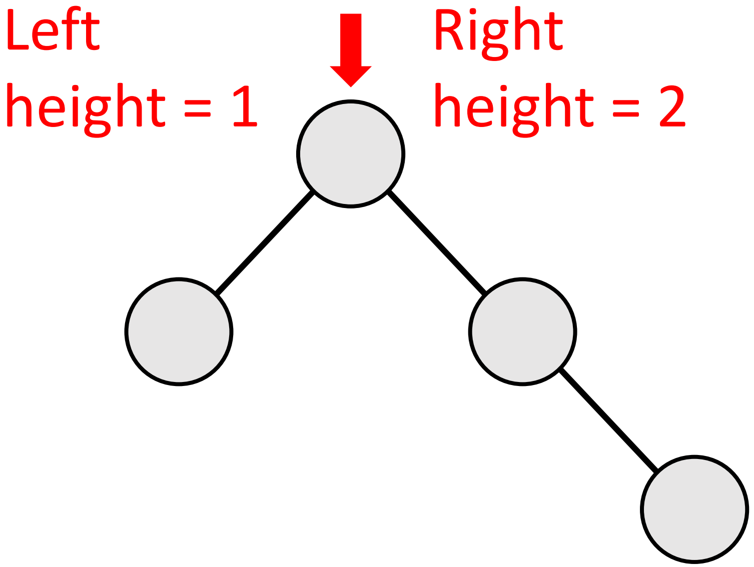
AVL trees

- AVL = Adelson-Velskii & Landis
- Insert node and keep track of height of subtrees of every node
  - Balance node every time difference between subtree heights is > 1
  - Basic balancing operation is **AVL rotation**

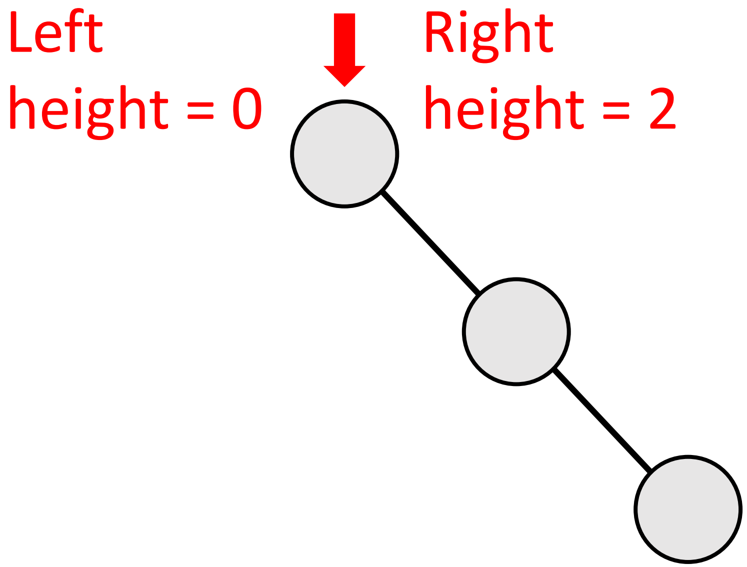
Adelson-Velskii, G.; E. M. Landis (1962). "An algorithm for the organization of information". *Proceedings of the USSR Academy of Sciences* **146**: 263–266. (Russian) English translation by Myron J. Ricci in *Soviet Math. Doklady* **3**:1259–1263, 1962.

AVL condition

- Is this node balanced?



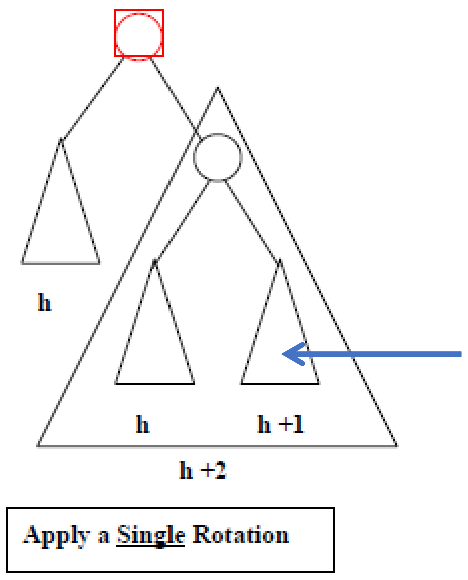
Balanced:
Difference is <=1



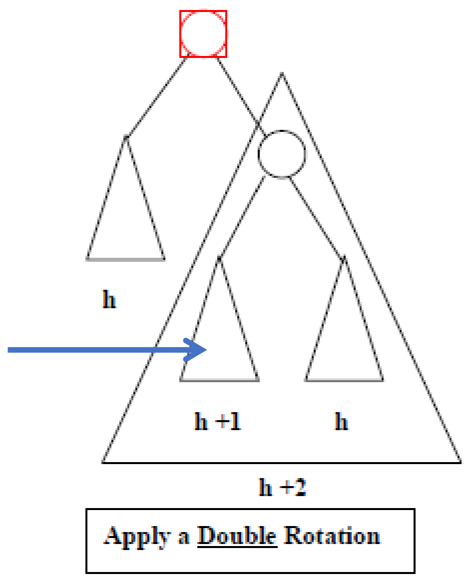
Unbalanced:
Difference is >1

Non-AVL tree caused by...

Outside insertion

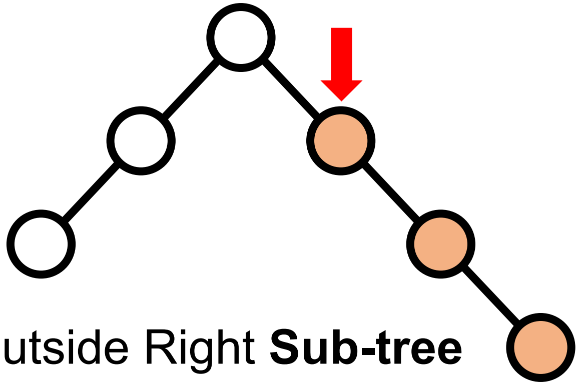


Inside insertion

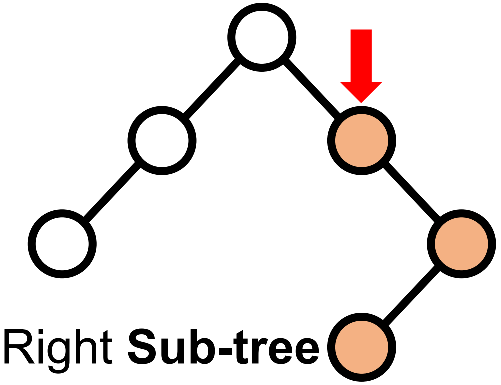


Mirror-symmetrical case is handled identically

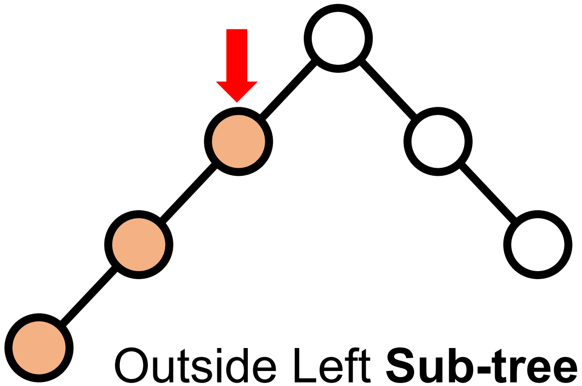
Unbalanced tree categories



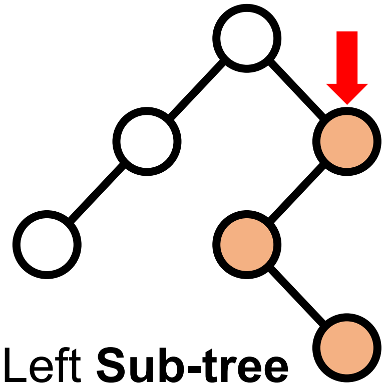
Outside Right **Sub-tree**
Right **Child** (RR)



Inside Right **Sub-tree**
Left **Child** (RL)

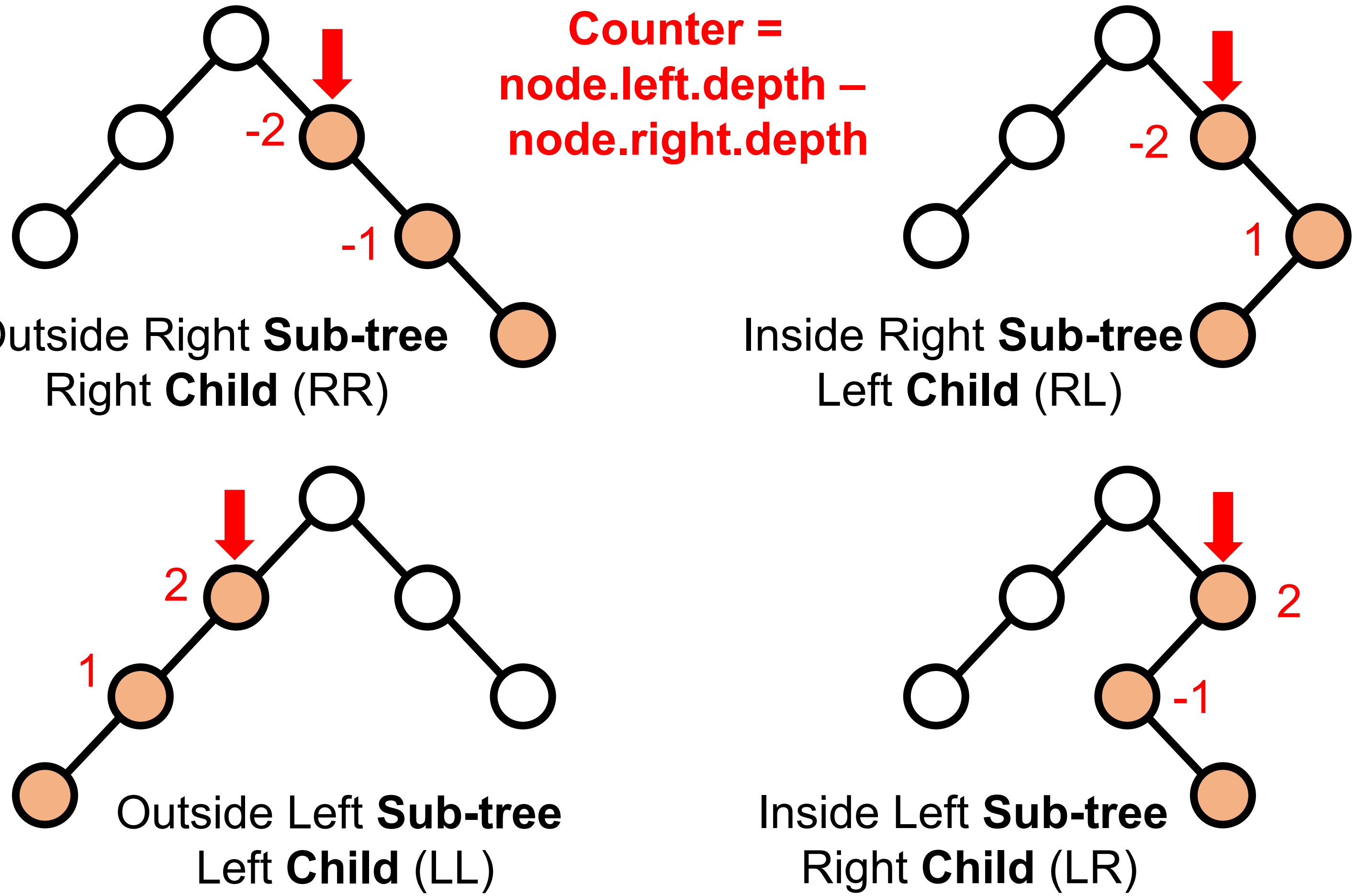


Outside Left **Sub-tree**
Left **Child** (LL)



Inside Left **Sub-tree**
Right **Child** (LR)

Unbalanced tree categories

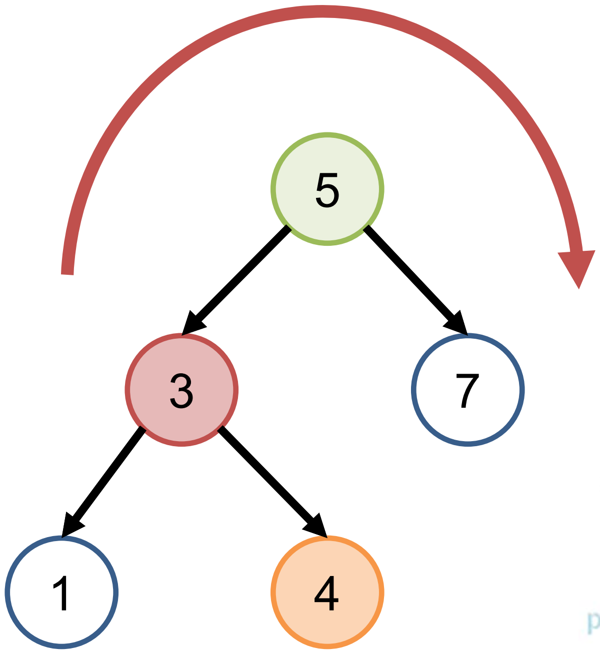


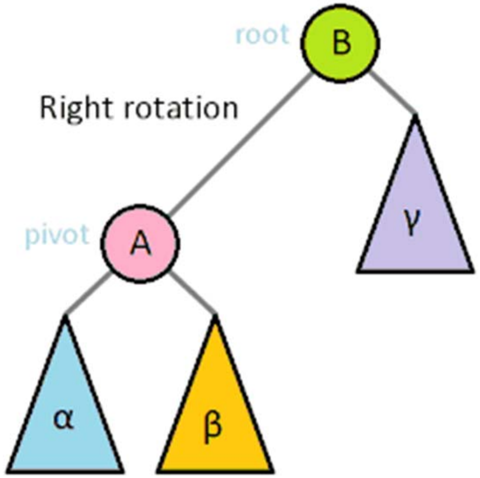
How to balance?

- Outside imbalance:
  - Rotate to rebalance
  - E.g., left subtree > right subtree: rotate right
- Inside imbalance:
  - Similar, but requires two rotations

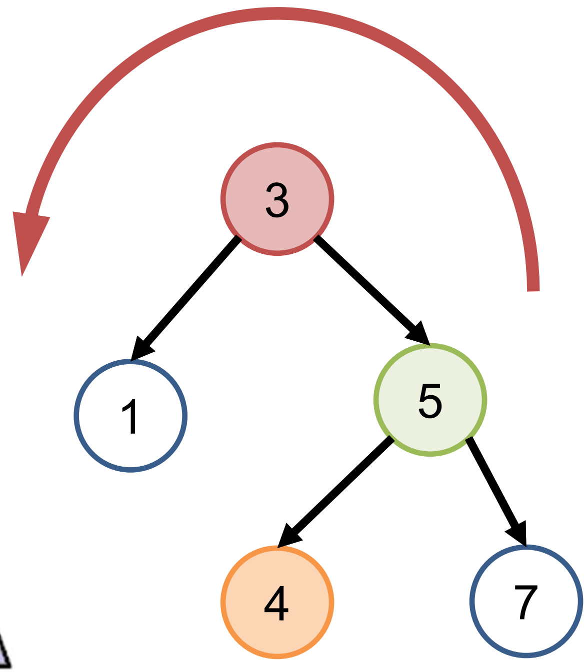
AVL rotation

Right Rotation

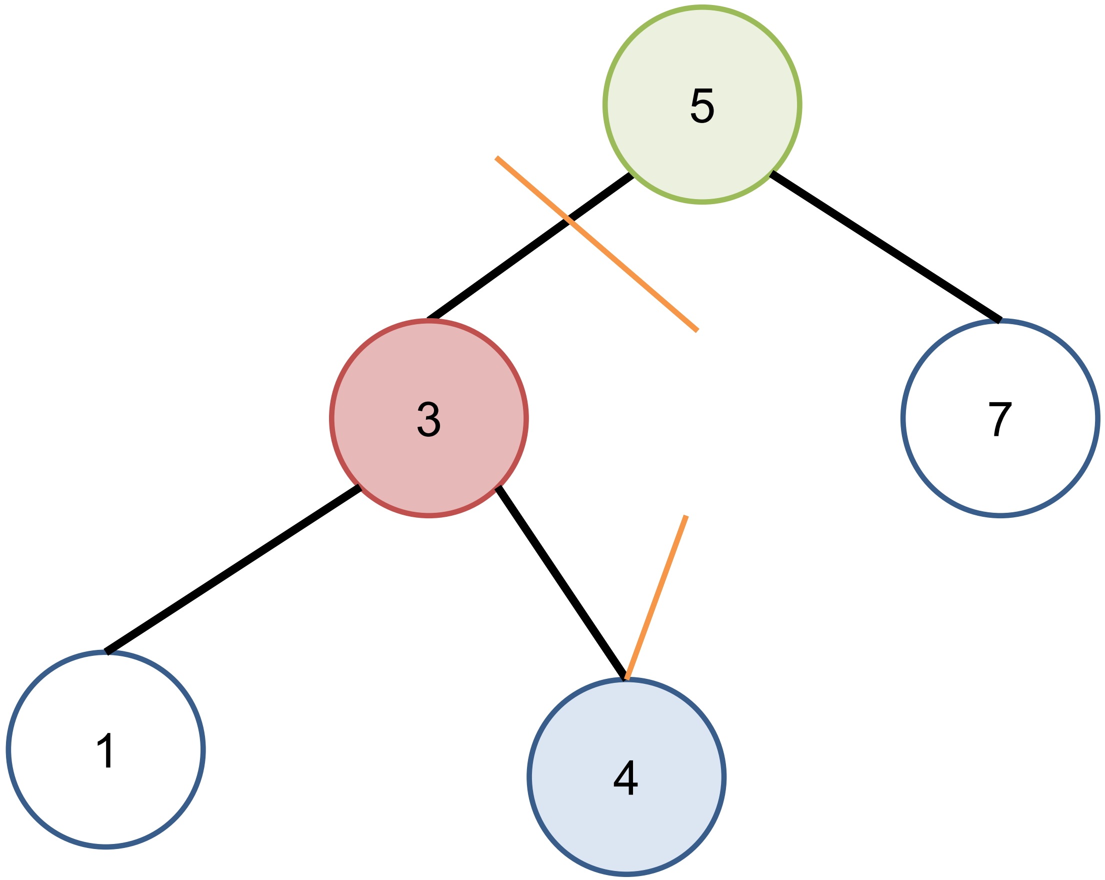




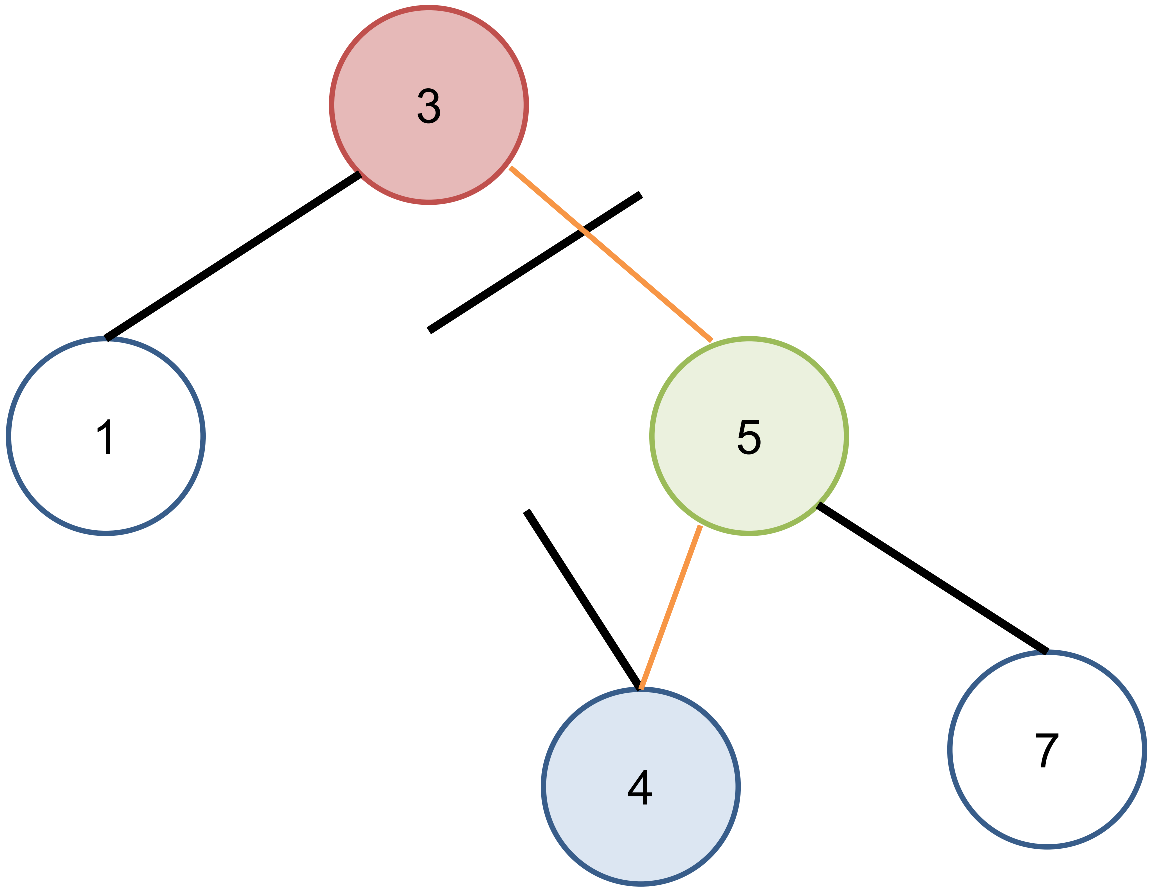
Left Rotation



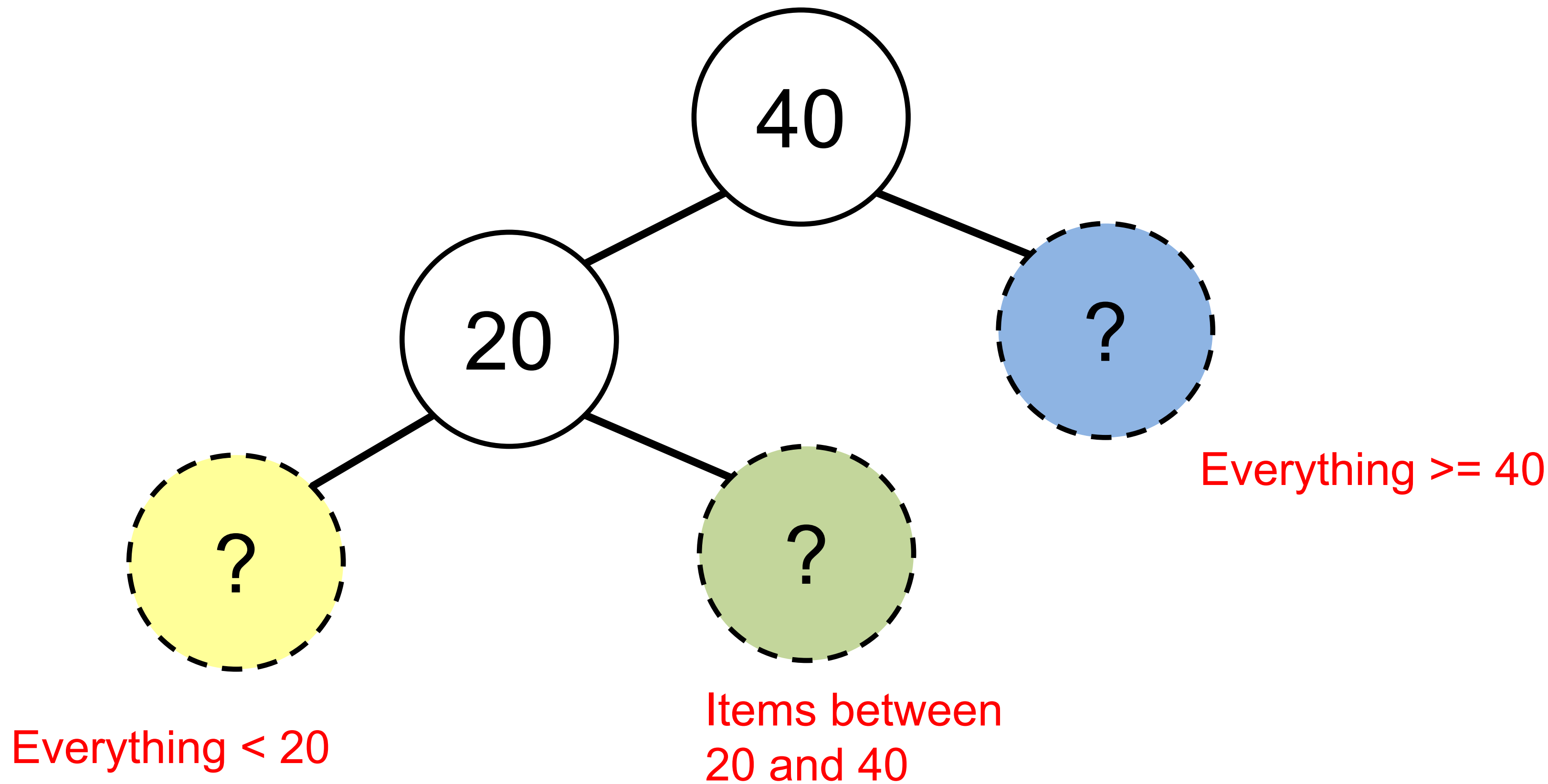
Right rotation



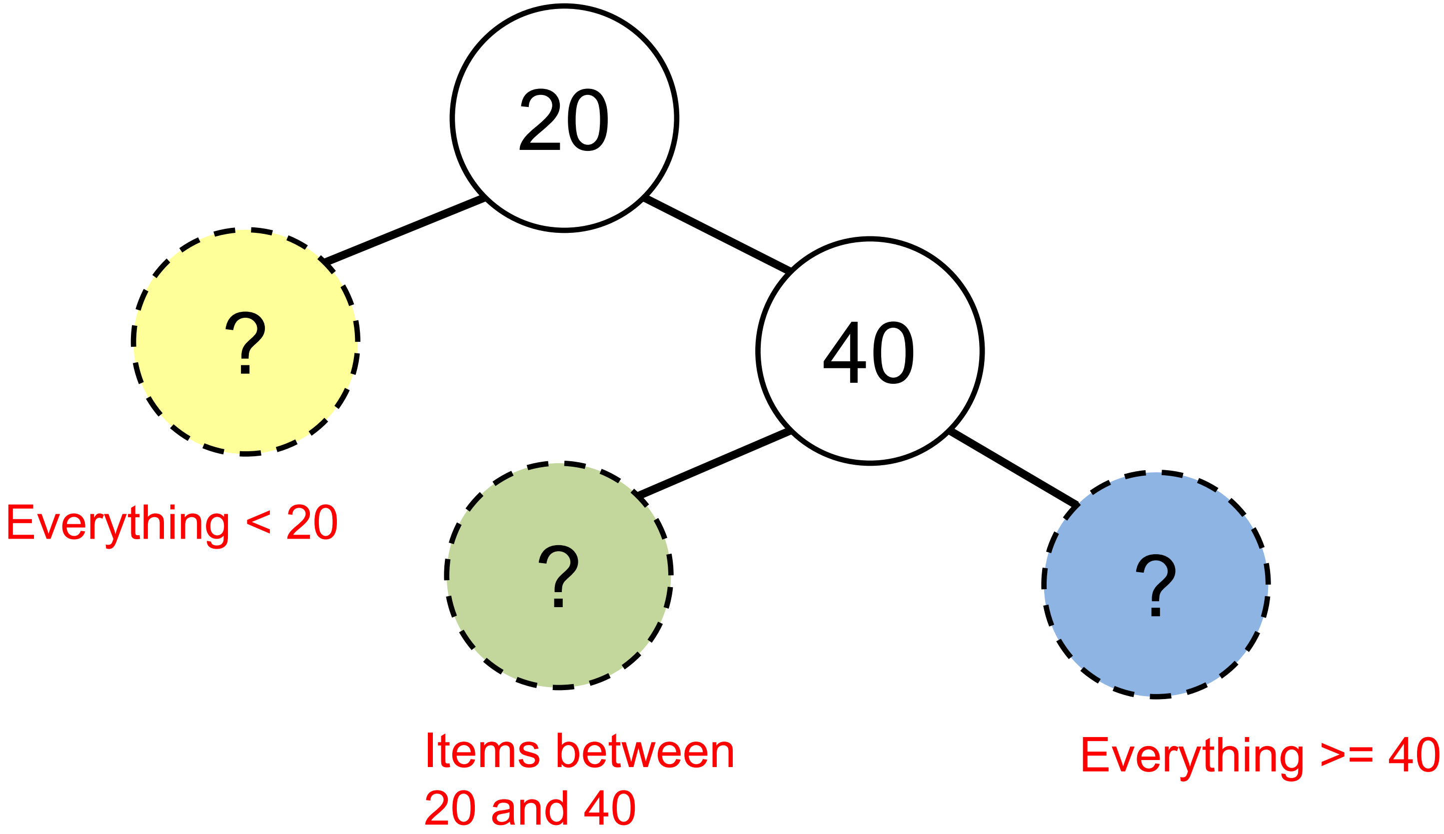
Left rotation

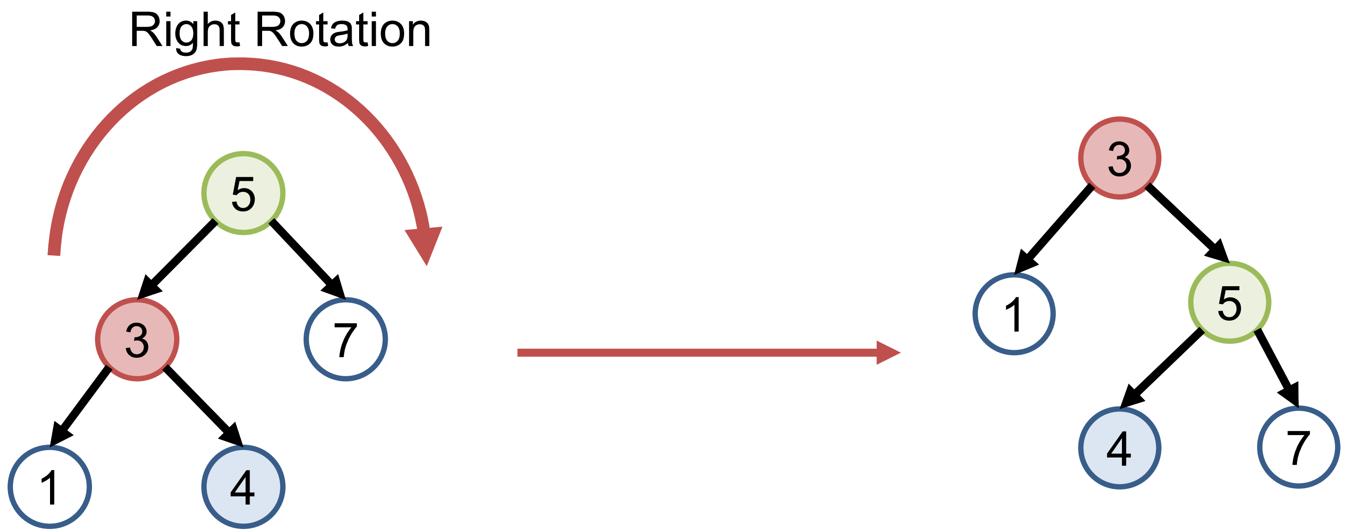


Preserving sorted order



Preserving sorted order



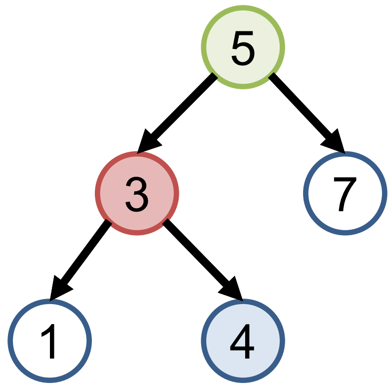


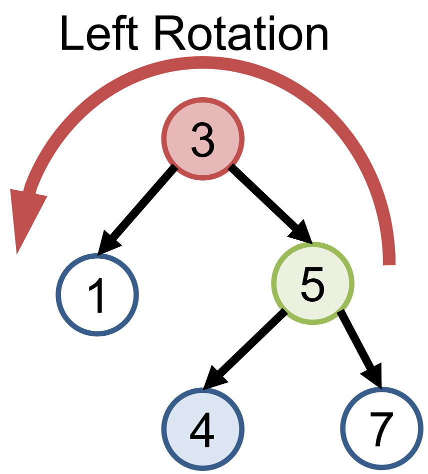

```
RotateR(node)
{
    //Auxiliary variables
    left = node.Left;
    leftRight = left.Right;
    parent = node.Parent;

    //Operations
    left.Parent = parent;
    left.Right = node;
    node.Left = leftRight;
    node.Parent = left;
}
```

```
RotateR(5)
{
    //Auxiliary variables
    left = 3;
    leftRight = 4;
    parent = Null;

    //Operations
    3.Parent = Null;
    3.Right = 5;
    5.Left = 4;
    5.Parent = 3;
}
```





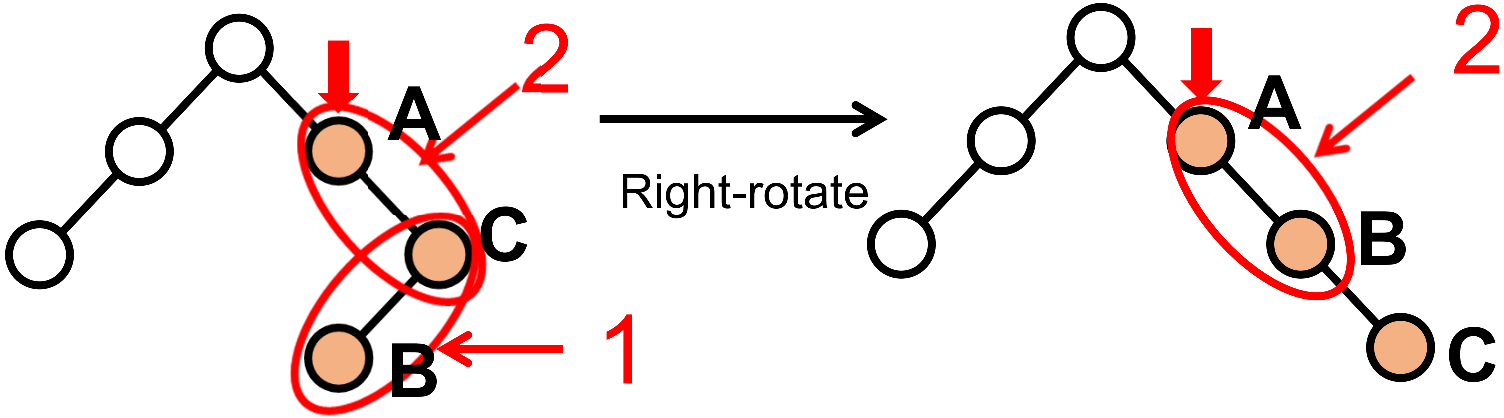

```
RotateL(node)
{
    //Auxiliary variables
    right = node.Right;
    rightLeft = right.Light;
    parent = node.Parent;

    //Operations
    right.Parent = parent;
    right.Left = node;
    node.Right = rightLeft;
    node.Parent = right;
}
```

```
RotateL(3)
{
    //Auxiliary variables
    right = 5;
    rightLeft = 4;
    parent = Null;

    //Operations
    5.Parent = Null;
    5.Left = 3;
    3.Right = 4;
    3.Parent = 5;
}
```

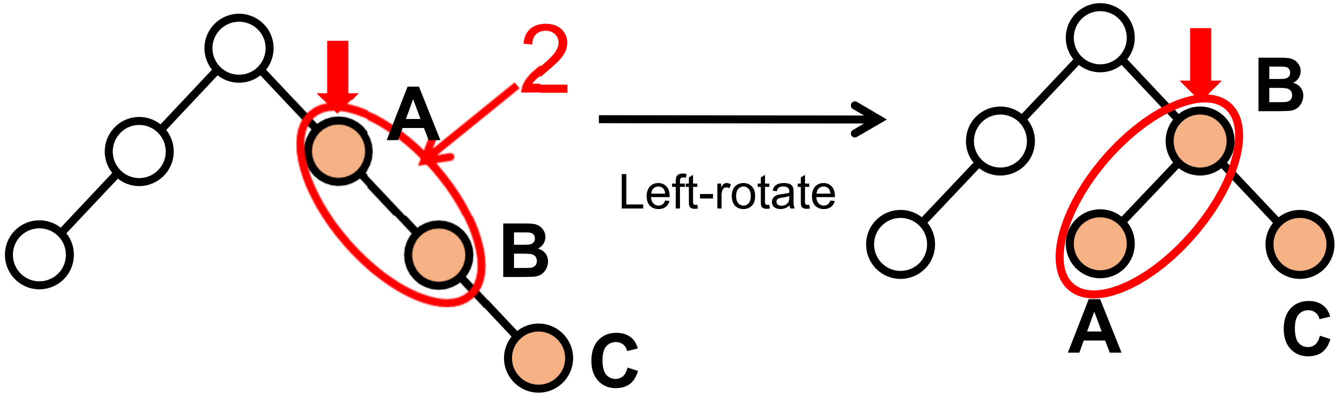
Inside imbalance: double rotation



Right Left (RL) double rotation:

- First rotation swaps grandchild and child (Right rotation)

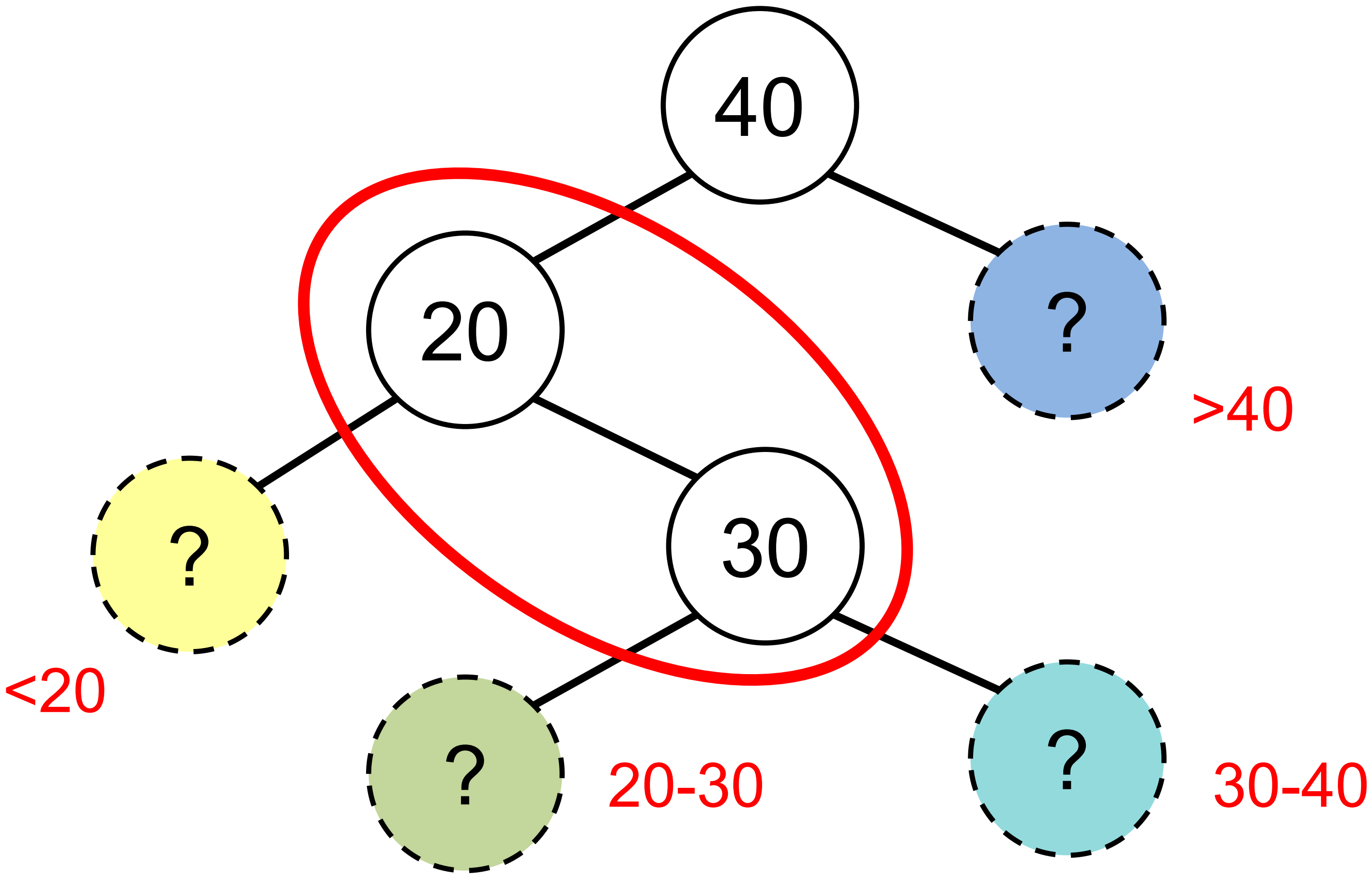
Inside imbalance: double rotation



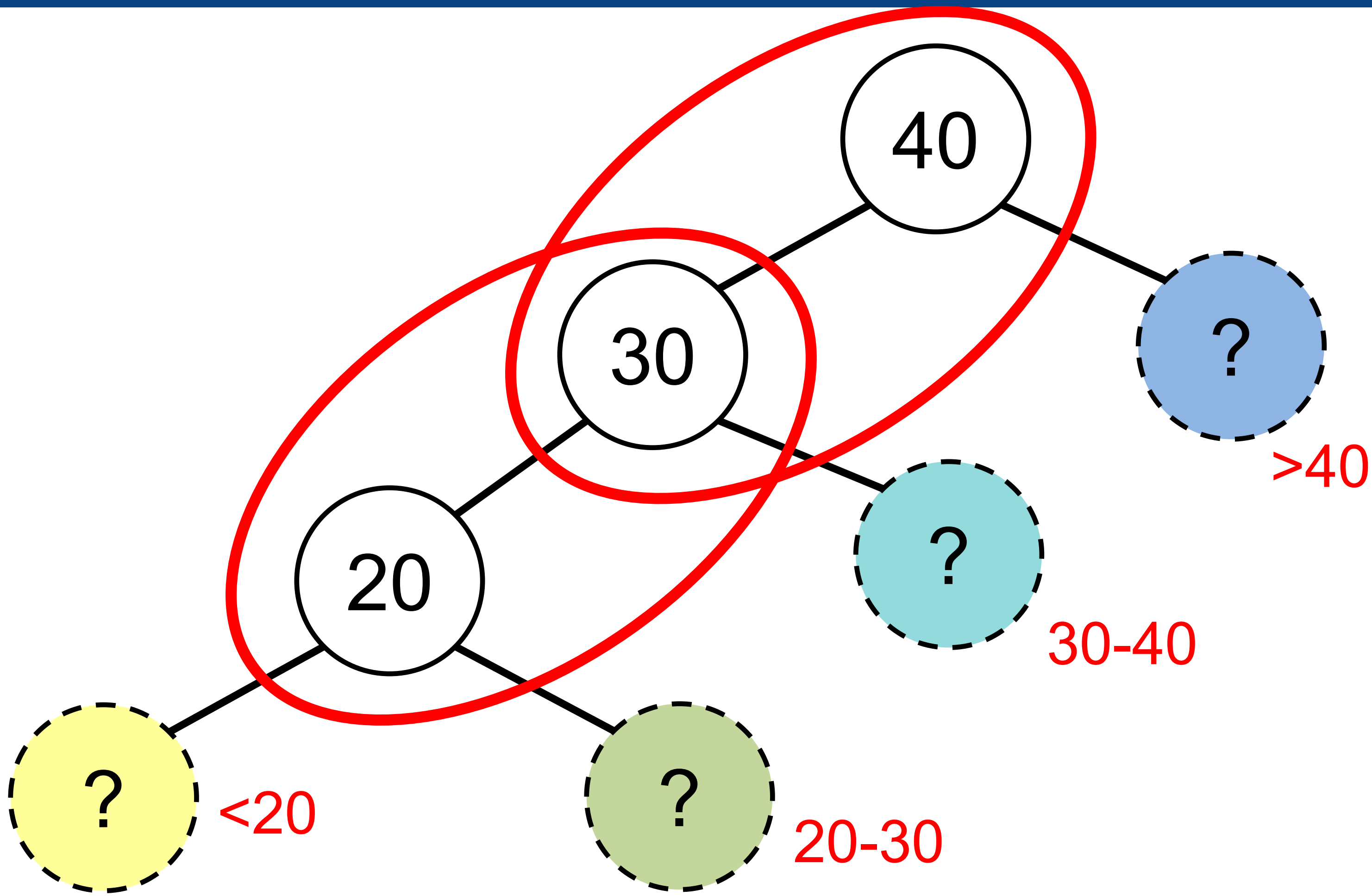
Right Left (RL) double rotation:

- Second rotation swaps parent and child (Left rotation)

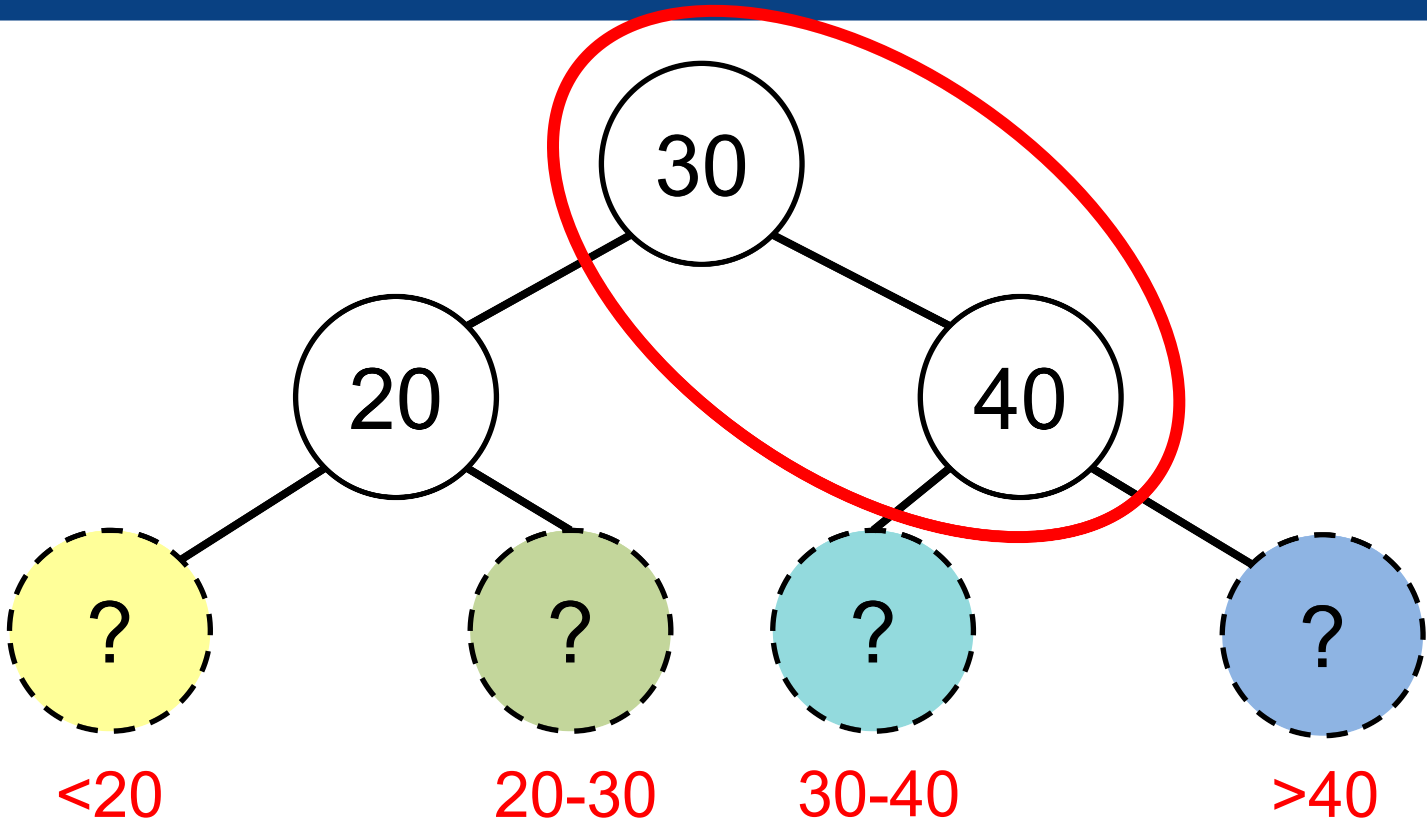
Preserving sorted order



Preserving sorted order



Preserving sorted order



AVL insertion

```
node* insert ( node* tree, node* new_node )
{
    if ( tree == NULL )
        tree = new_node;
    else if ( new_node->key < tree->key ) {
        tree->left = insert ( tree->left, new_node );
        /* Fifty lines of left balancing code */
    }
    else {
        tree->right = insert ( tree->right, new_node );
        /* Fifty lines of right balancing code */
    }
    return tree;
}
```

Summary: AVL trees

- Good features:
  - Tree is always nearly balanced
  - Actually, height < 1.44 $\log_2(n)$
  - Therefore complexity for any search is O(log n)

- Less ideal features:
  - Very fiddly to code, must keep track of
    - insertion path
    - height of all subtrees
  - Balancing adds time to insertion (but constant time)