



香港中文大學 (深圳)  
The Chinese University of Hong Kong

# CSC3100 Data Structures

## Lecture 15: AVL tree

Yixiang Fang  
School of Data Science (SDS)  
The Chinese University of Hong Kong, Shenzhen

---



# Outline

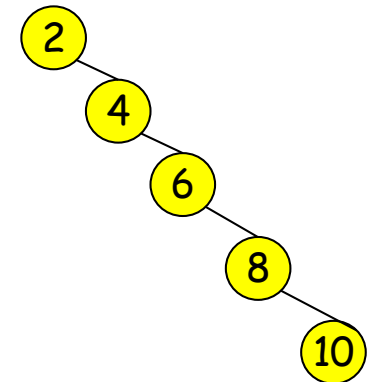
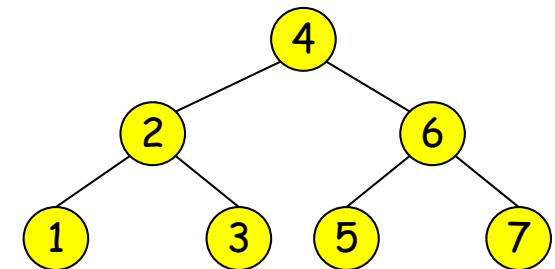
---

- ▶ AVL tree
  - Motivation
  - Formal definition
  - Insertion, rebalance strategies, deletion
  - Examples



# Analysis of binary search tree

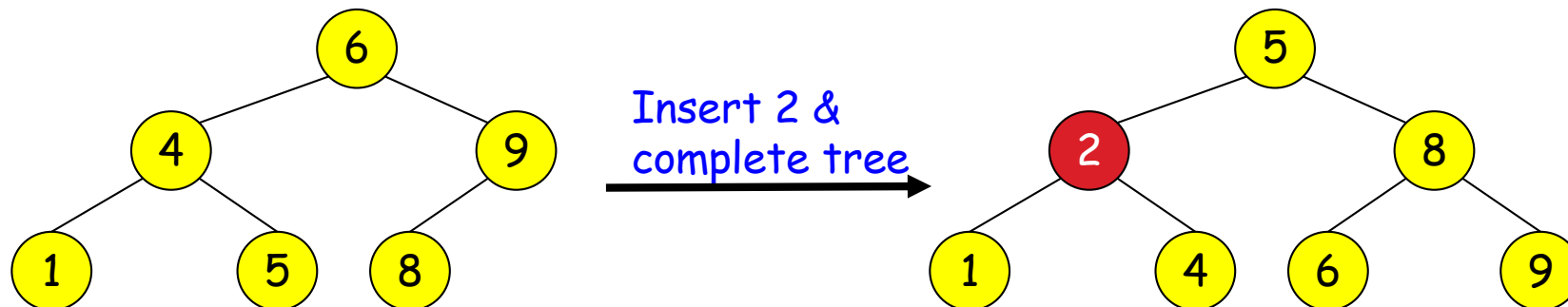
- ▶ All BST operations need  $O(d)$  time cost, where  $d$  is the tree depth and  $\log n \leq d \leq n-1$ 
  - Thus, they take at most  $O(n)$  time and at least  $O(\log n)$  time
- ▶ What is the best-case tree?
  - A complete binary tree
- ▶ What is the worst-case tree?
  - All nodes form a chain
  - E.g., inserting 2, 4, 6, 8, 10 into an empty BST





# Balanced binary search tree

- ▶ Want a **complete tree** after every operation
  - A **complete** binary tree has height of  $O(\log n)$
  - The tree is full except possibly in the lower right
- ▶ This is expensive
  - For example, insert 2 in the tree on the left and then rebuild as a complete tree



- Solution: we relax the condition a little bit -> balanced BST



# Balanced binary search tree

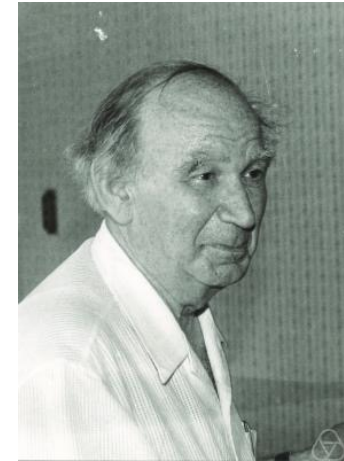
---

- ▶ Unless keys appear in just the right order, imbalance will occur on the updated BST
  - In fact, the order of keys defines the structure of the tree
- ▶ Many algorithms exist for keeping binary search trees balanced
  - AVL trees
  - Red-black trees
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees



# AVL tree

- ▶ Invented in 1962 by
  - Georgy Adelson-Velsky
  - Evgenii Landis

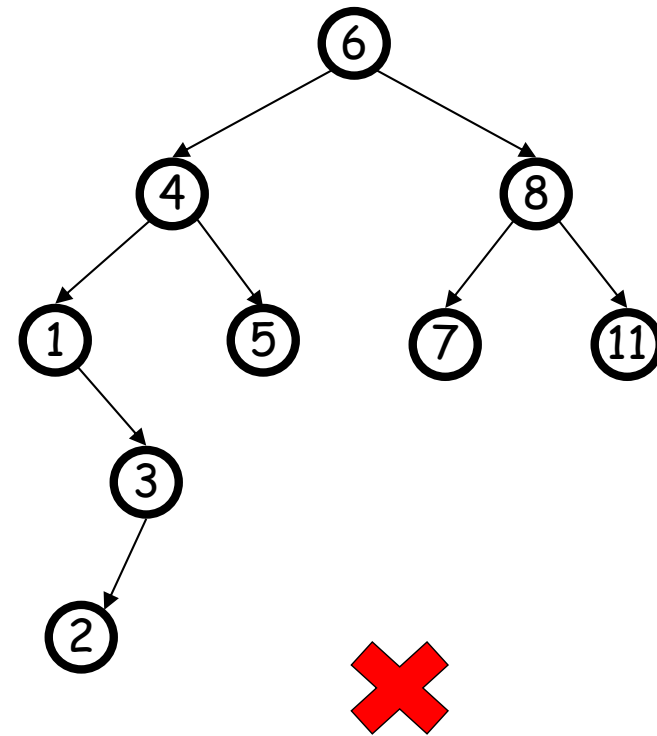
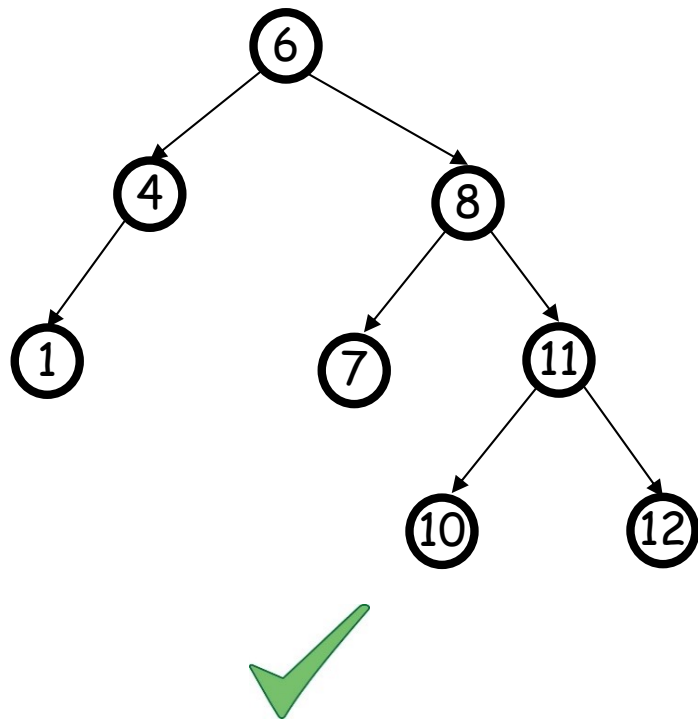


- ▶ An AVL tree is a self-balancing BST s.t.
  - For every node in the tree, the height of the left subtree differs from the height of the right subtree by at most 1
    - Balance factor of a node:  $\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})$
  - If at any time they differ by more than one, rebalancing is done to restore this property



# AVL tree examples

Balance condition: balance factor of every node is between -1 and 1





# AVL tree properties

## ▶ Structural properties

- Binary tree property (same as for BST)
- Order property (same as for BST)

Balance condition: balance factor of every node is between -1 and 1, where  $\text{balance}(\text{node}) = \text{height}(\text{node.left}) - \text{height}(\text{node.right})$

- An empty tree has a height of -1, and leaves have a height of 0

The height of a node is the length of the longest path from it to a leaf

## ▶ The worst-case depth is $O(\log n)$

- All operations depend on the depth of the tree
- Find, insertion, and deletion can be completed in  $O(\log n)$ , where  $n$  is the number of nodes in the tree



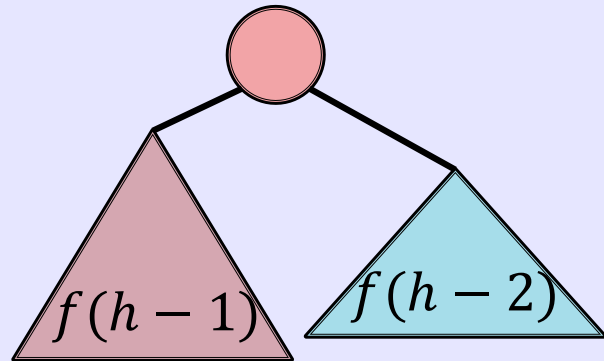


# Height of binary search tree

Theorem 1: Given a balanced binary search tree  $T$  of  $n$  nodes, the height, or equivalently the depth, of  $T$  is  $O(\log n)$ .

Proof: Let  $f(h)$  be the minimum number of nodes of a balanced BST of height  $h$ . Then, it is easy to verify that  $f(1) = 2, f(2) = 4, f(2) = 7$ .

For any  $h \geq 3$ , we have that  $f(h) = f(h-1) + f(h-2) + 1$



$$\begin{aligned} f(h) &> f(h-1) + f(h-2) \\ &> 2f(h-2) \\ &> 4f(h-4) \\ &\dots \\ &> 2^{\frac{h}{2}-1} \cdot f(2) = 2^{\frac{h}{2}} \end{aligned}$$

Therefore, given a balanced BST of  $n$  nodes of height  $h$ , we have:

$$n > 2^{\frac{h-1}{2}} \Rightarrow h < 2 \log_2 n + 1 \Rightarrow h = O(\log n)$$



# Insertion on AVL tree

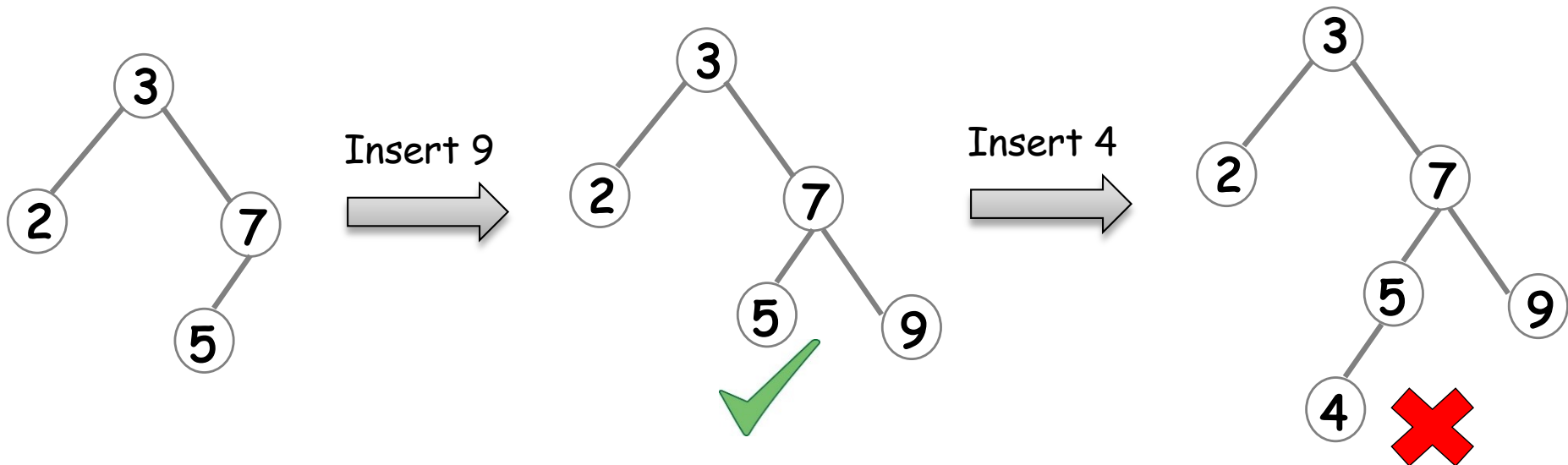
---

- ▶ Insertion at the leaf may cause unbalance
- ▶ Two principles:
  - Imbalance will only occur on **the path from the inserted node to the root** (only these nodes have had their subtrees altered - local problem)
  - Rebalancing should occur at the deepest unbalanced node (local solution too)
- ▶ General idea of rebalancing
  - After the insertion, **go back up** to the root node by node, updating heights
  - If a new balance factor is 2 or -2, rebalance tree by **rotation** around the node



# Insertion on AVL tree

- ▶ Steps of insertion:
  - Search for the element
  - If it is not there, insert it in its place



- ▶ Rebalance strategies:
  - **Rotation** allows us to change the structure without violating the BST property



# Insertion on AVL tree

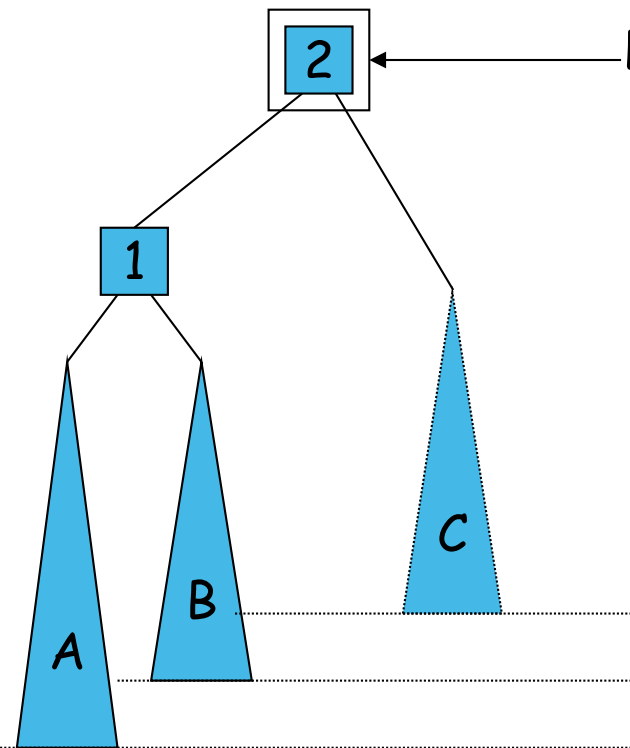
---

- ▶ There are 4 cases:
  - Outside cases (require single rotation) :
    - Left-left: insertion into **left** subtree of **left** child
    - Right-right: Insertion into **right** subtree of **right** child
    - (These two cases are symmetry)
  - Inside cases (require double rotation) :
    - Left-right: insertion into **left** child's **right** subtree
    - Right-left: insertion into **right** child's **left** subtree
    - (These two cases are symmetry)

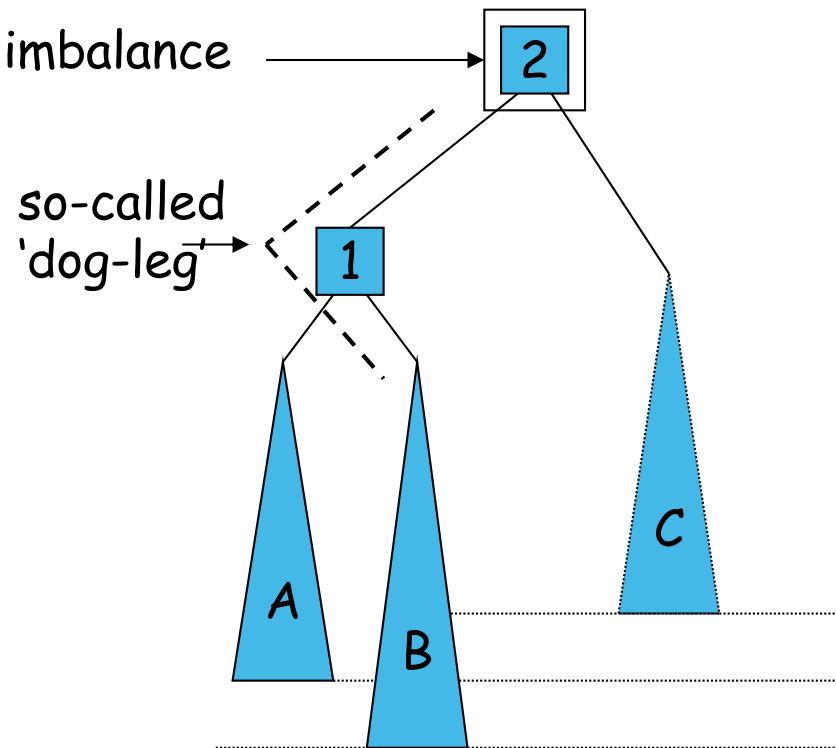


# AVL tree: resolving imbalance issue

- ▶ Left-left (right-right)



- ▶ Left-right (right-left)



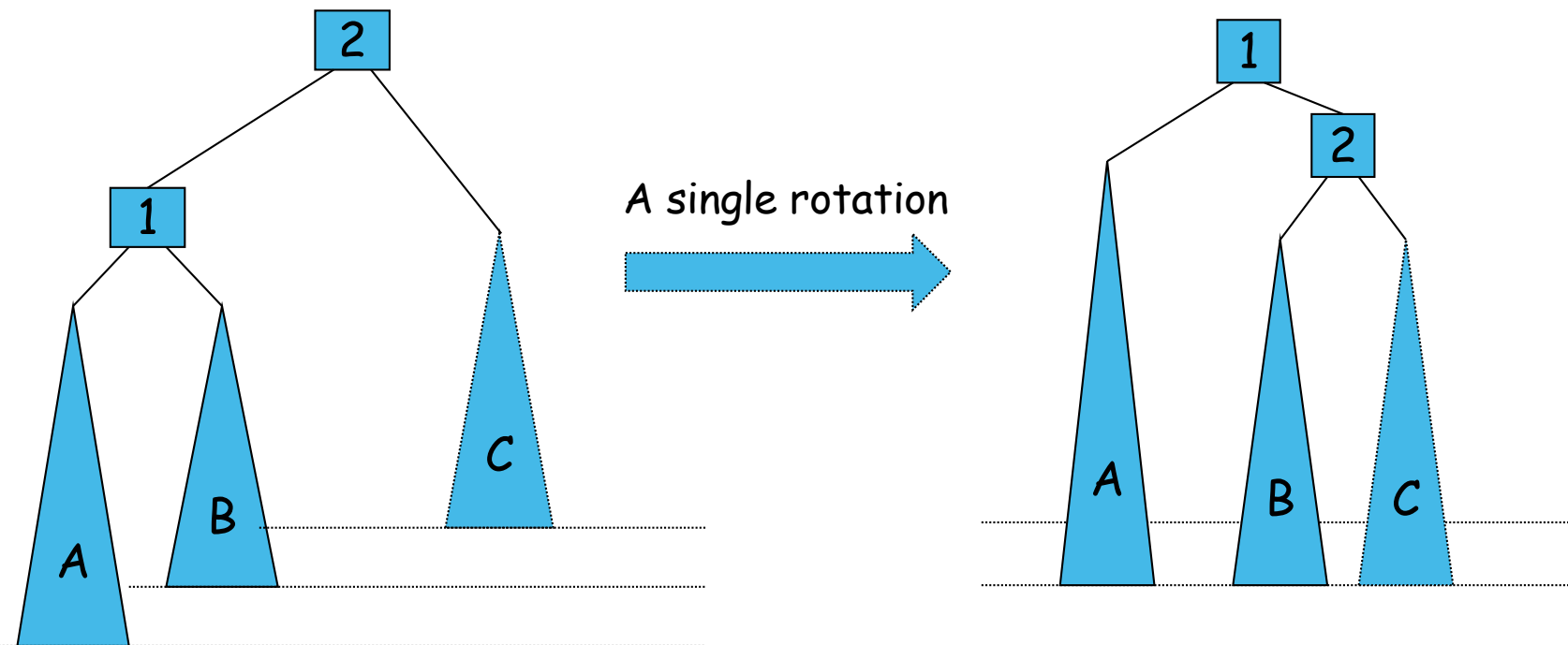
There are no other possibilities for the left (or right) subtree



# Left-left imbalance

[and right-right imbalance, by symmetry]

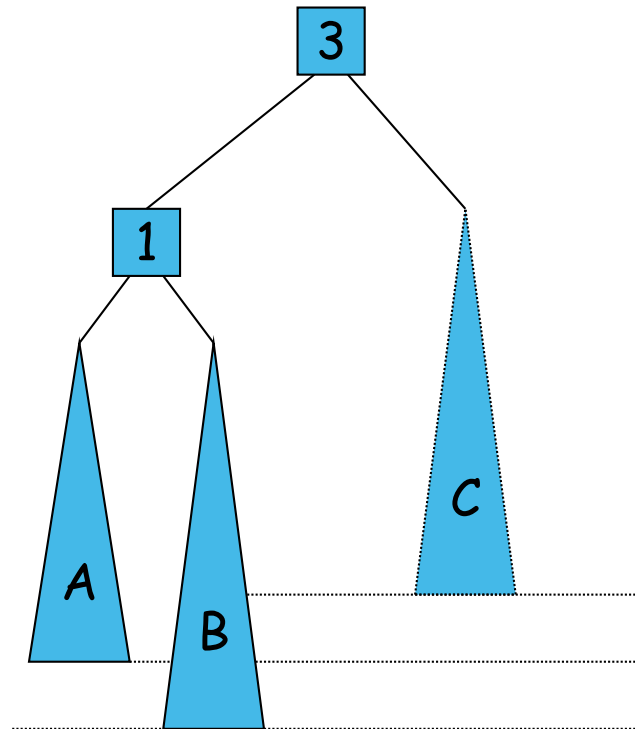
- ▶ Subtrees B and C have the same height
- ▶ Subtree A is one level higher
- ▶ Therefore, make 1 the new root, 2 its right child and B and C the subtrees of 2





# Left-right imbalance [and right-left imbalance by symmetry]

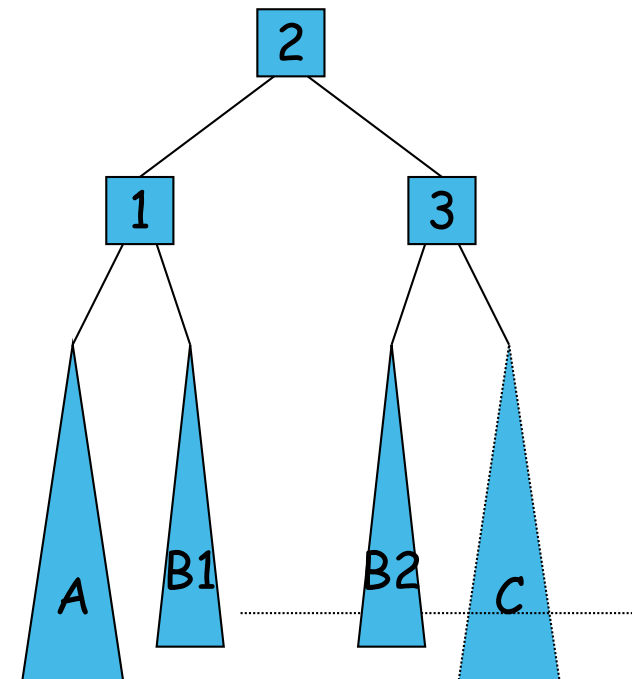
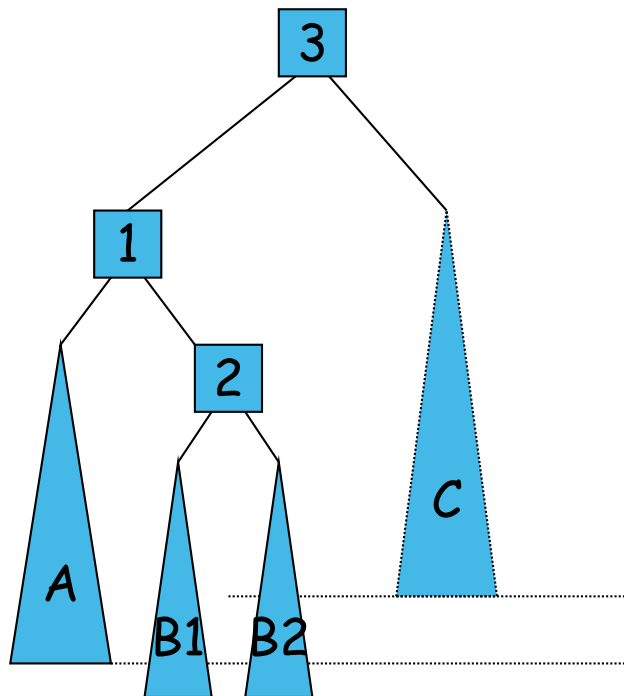
- ▶ Can't use the left-left balance trick
  - because now it's the middle subtree, i.e., B, that's too deep
- ▶ Instead consider what's inside B...





# Left-right imbalance [and right-left imbalance by symmetry]

- ▶ B will have two subtrees having at least one item
- ▶ We do not know which is too deep - set them both to 0.5 levels below subtree A
- ▶ Neither 1 nor 3 worked as root node, so make 2 the root
- ▶ Rearrange the subtrees
- ▶ No matter how deep B1 or B2 (+/- 0.5 levels) we get a legal AVL tree again

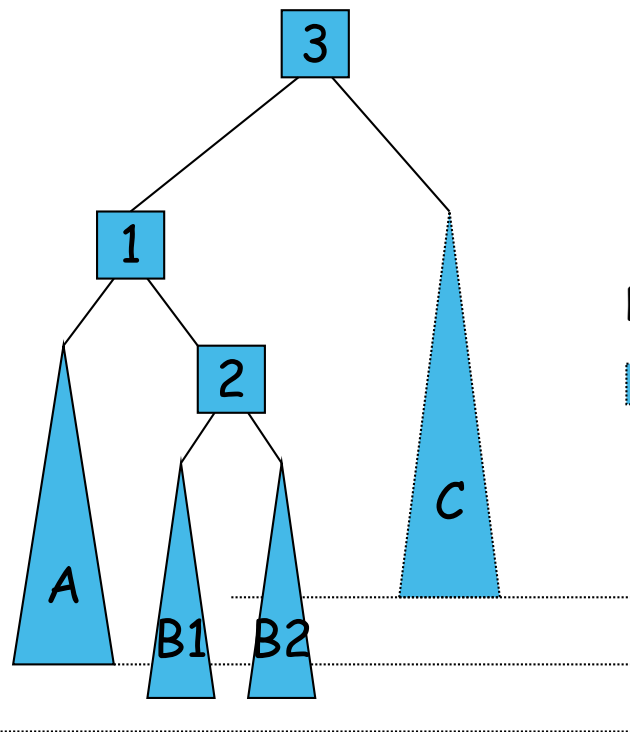




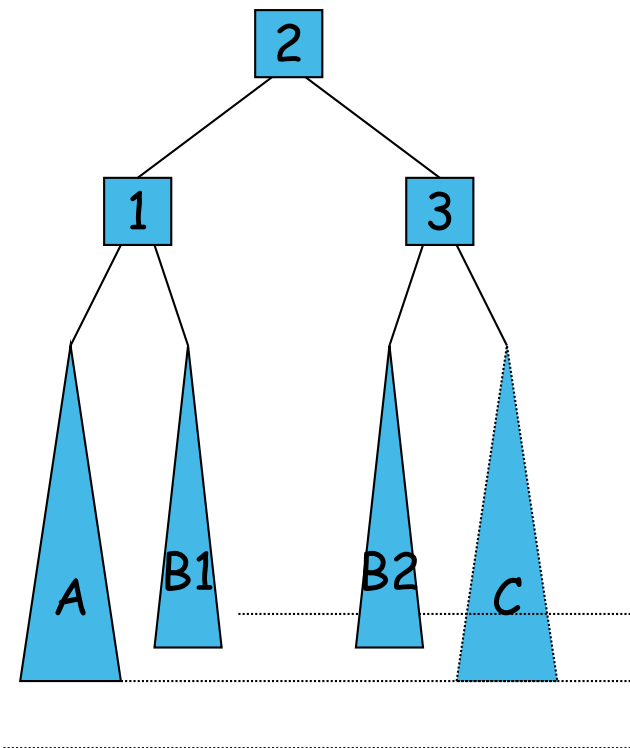


# Left-right imbalance [and right-left imbalance by symmetry]

## ► Double rotations

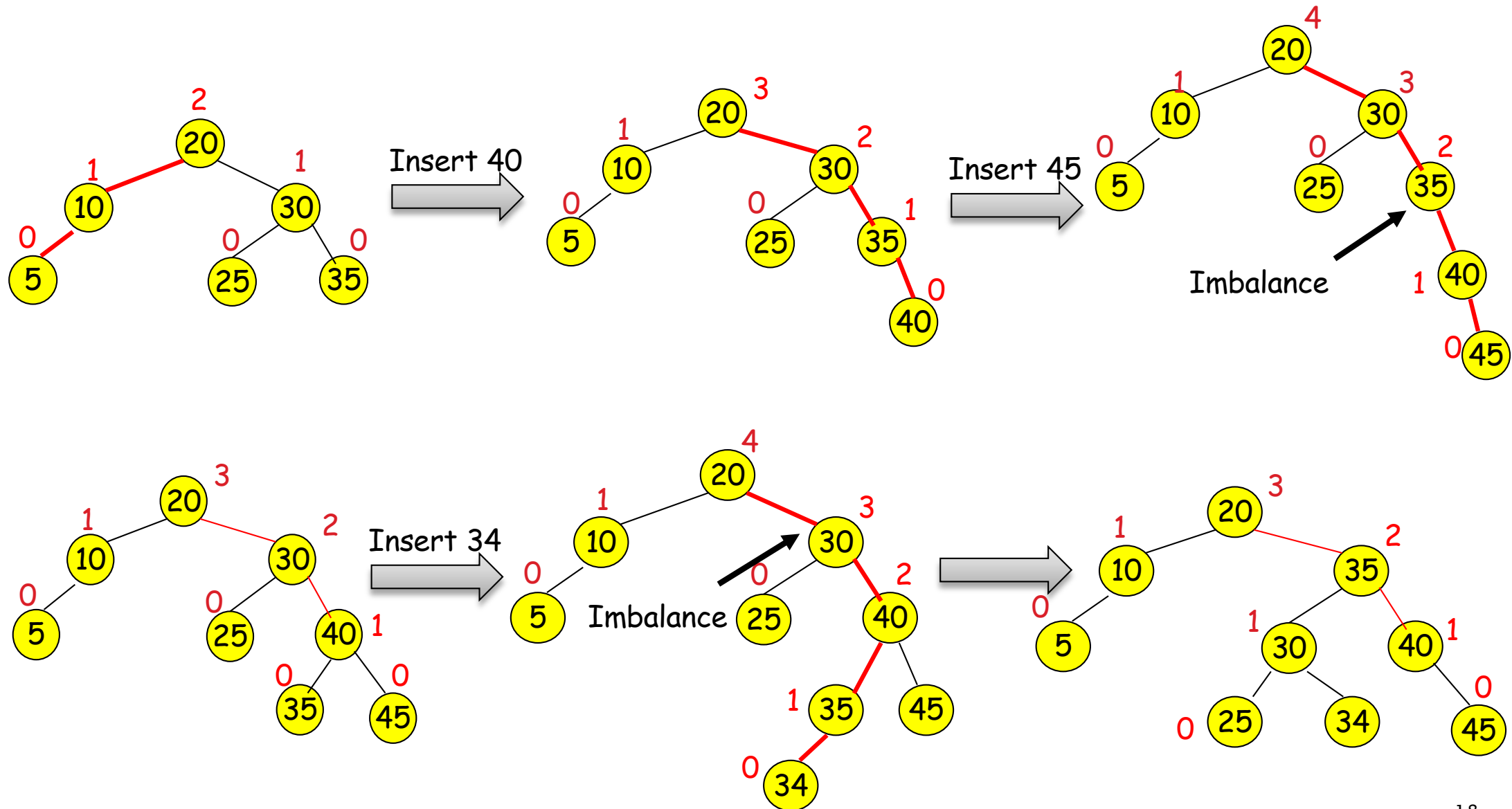


Double rotations





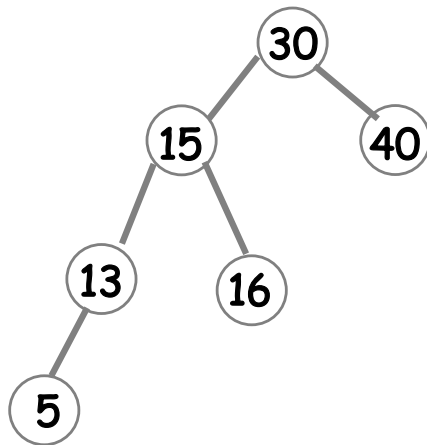
# Insertion examples



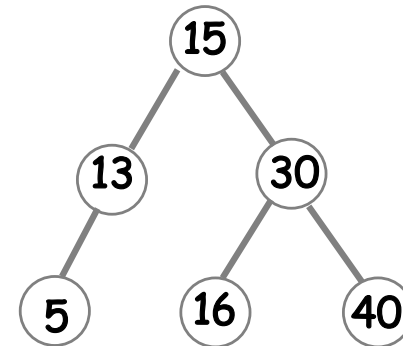


# Exercises

- ▶ Show the balanced BST after inserting key 5



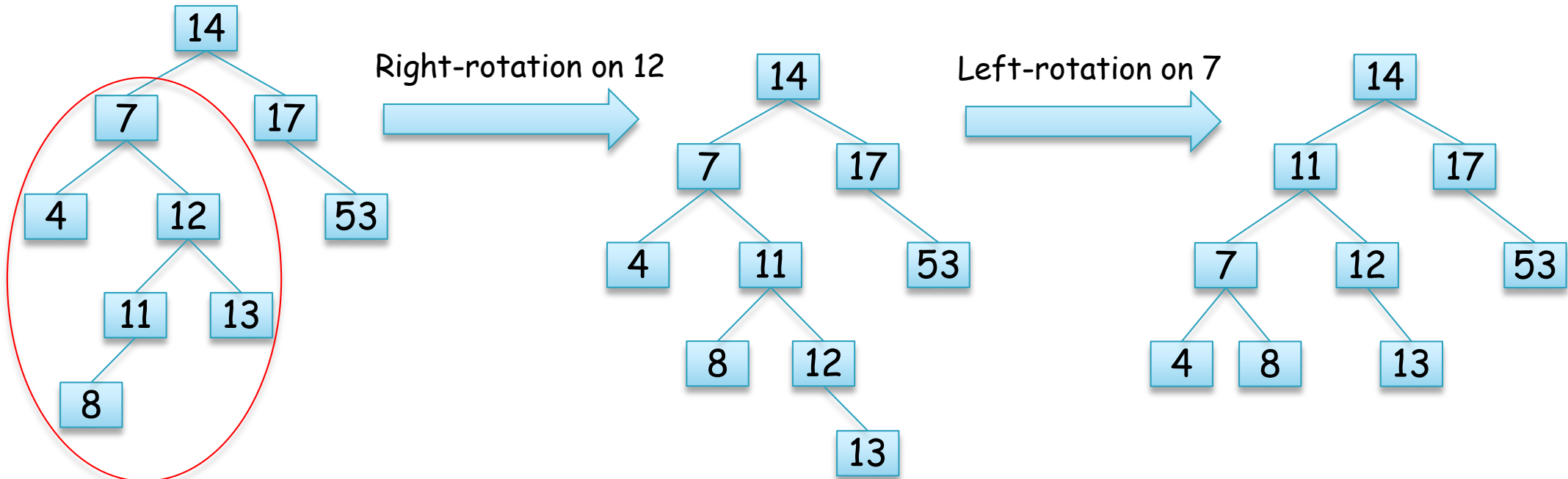
Right-rotation on 30





# Exercise

- Show the balanced BST after inserting key 8





# Rebalance implementation

```
private AvlNode<Anytype> insert(Anytype x, AvlNode<Anytype> t ) {
/*1*/    if( t == null )    t = new AvlNode<Anytype>( x, null, null );
/*2*/    else if( x.compareTo( t.element ) < 0 )
    {
        t.left = insert( x, t.left );
        if( height( t.left ) - height( t.right ) == 2 )
            if( x.compareTo( t.left.element ) < 0 )
                t = rotateWithLeftChild( t );
            else
                t = doubleWithLeftChild( t );
    }
/*3*/    else if( x.compareTo( t.element ) > 0 )
    {
        t.right = insert( x, t.right );
        if( height( t.right ) - height( t.left ) == 2 )
            if( x.compareTo( t.right.element ) > 0 )
                t = rotateWithRightChild( t );
            else
                t = doubleWithRightChild( t );
    }
/*4*/    else
        ; // Duplicate; do nothing
    t.height = max( height( t.left ), height( t.right ) ) + 1;
    return t;
}
```



# Rebalance implementation

---

```
private static AvlNode<Anytype> rotateWithLeftChild(AvlNode<Anytype> k2 )
{
    AvlNode<Anytype> k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    k2.height = max(height(k2.left), height(k2.right)) + 1;
    k1.height = max(height(k1.left), k2.height) + 1;
    return k1;
}
```

```
private static AvlNode<Anytype> rotateWithRightChild( AvlNode<Anytype> k1 )
{
    AvlNode<Anytype> k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    k1.height = max( height( k1.left ), height( k1.right ) ) + 1;
    k2.height = max( height( k2.right ), k1.height ) + 1;
    return k2;
}
```



# Rebalance implementation

---

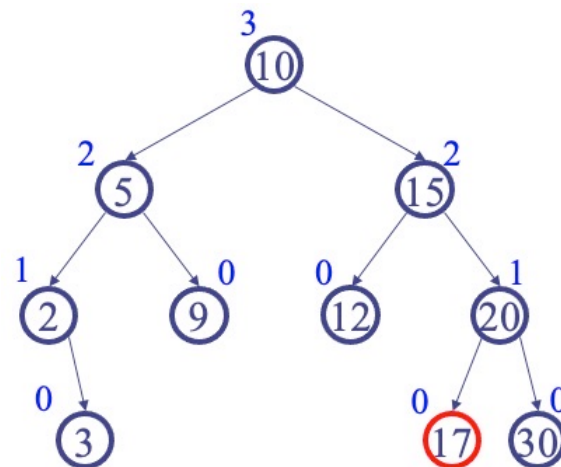
```
private static AvlNode<Anytype> doubleWithLeftChild( AvlNode<Anytype> k3 )
{
    k3.left = rotateWithRightChild( k3.left );
    return rotateWithLeftChild( k3 );
}
```

```
private static AvlNode<Anytype> doubleWithRightChild( AvlNode<Anytype> k1 )
{
    k1.right = rotateWithLeftChild( k1.right );
    return rotateWithRightChild( k1 );
}
```



# Deletion on AVL-tree

- ▶ Similar but more complex than insertion
  - Rotations and double rotations needed to rebalance
  - Imbalance may propagate upward so that many rotations may be needed
- ▶ Easy case: no rotation (Delete 17)

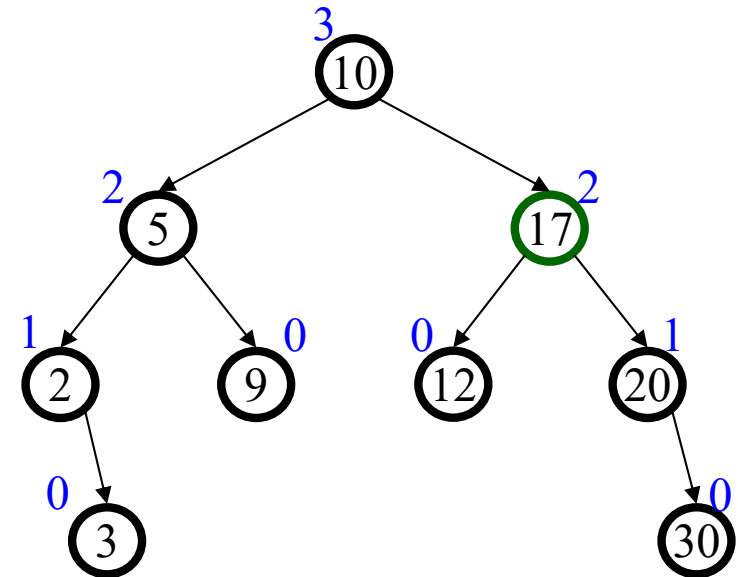
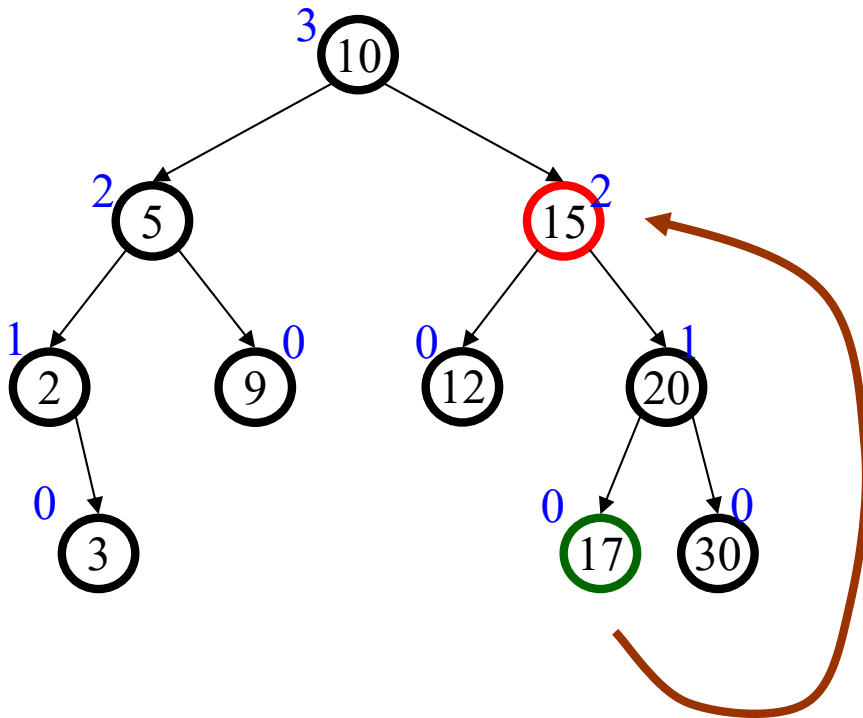






# Deletion on AVL-tree

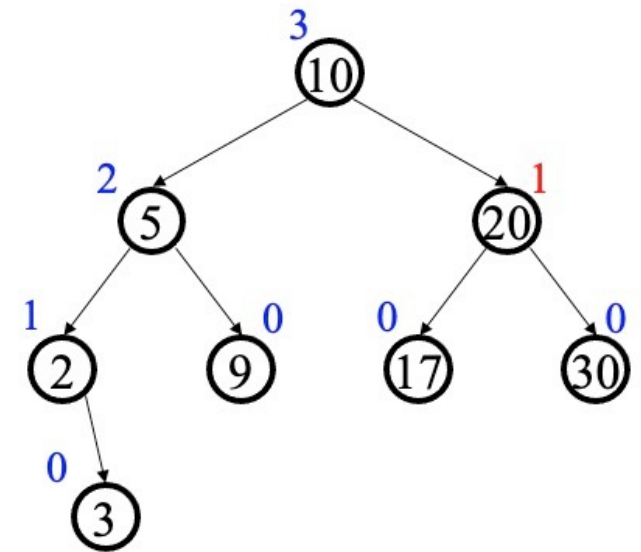
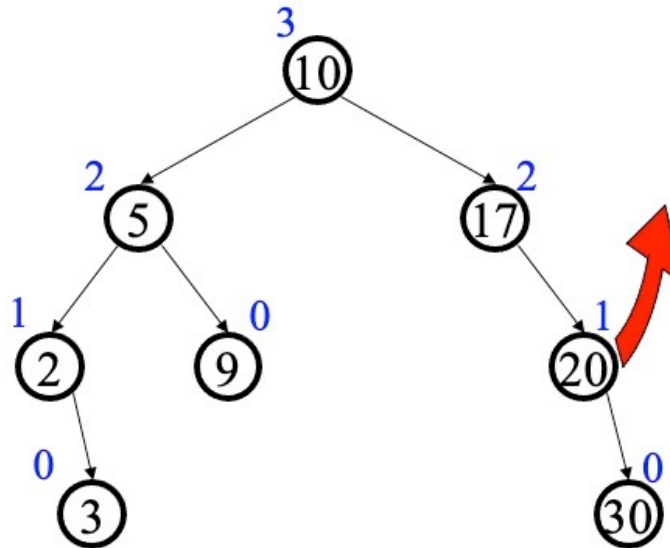
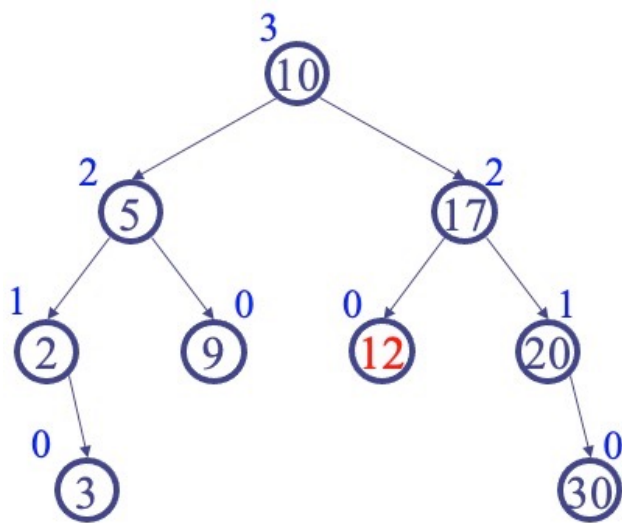
- ▶ Easy case: no rotation (Delete **15**)





# Deletion on AVL-tree

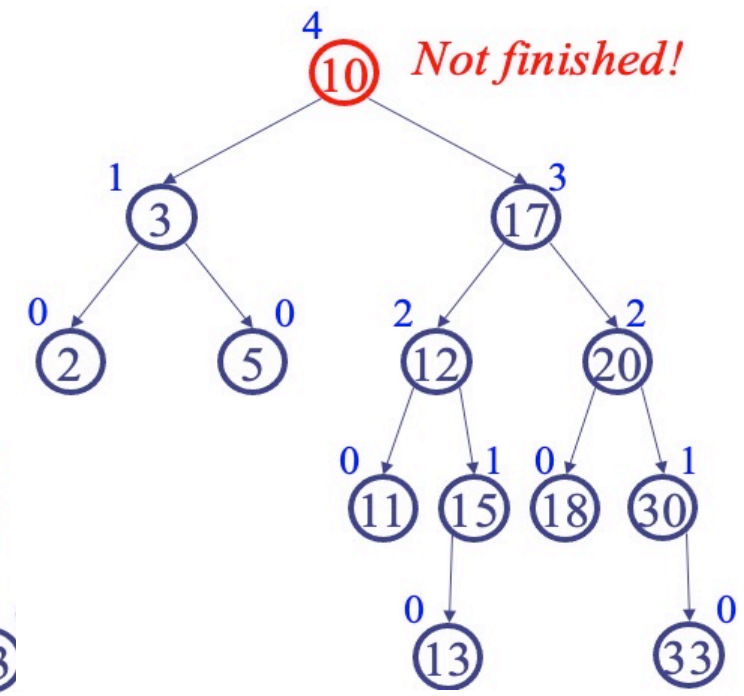
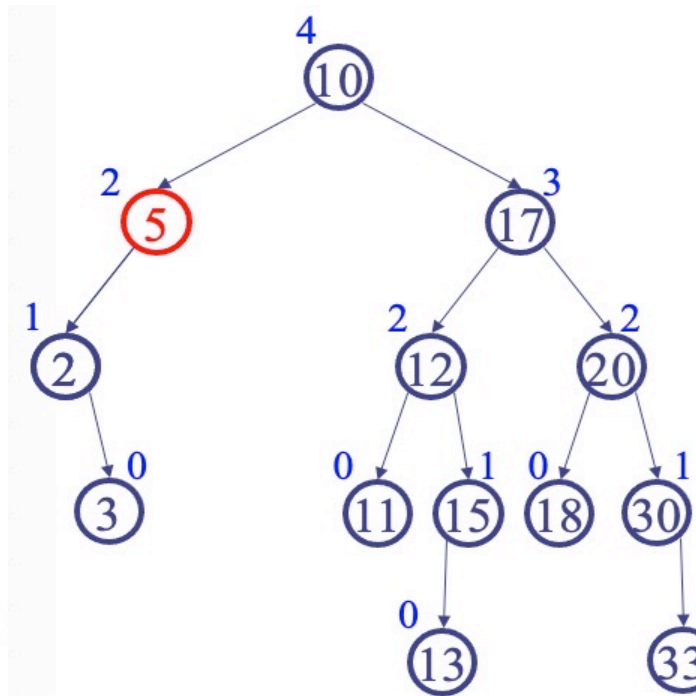
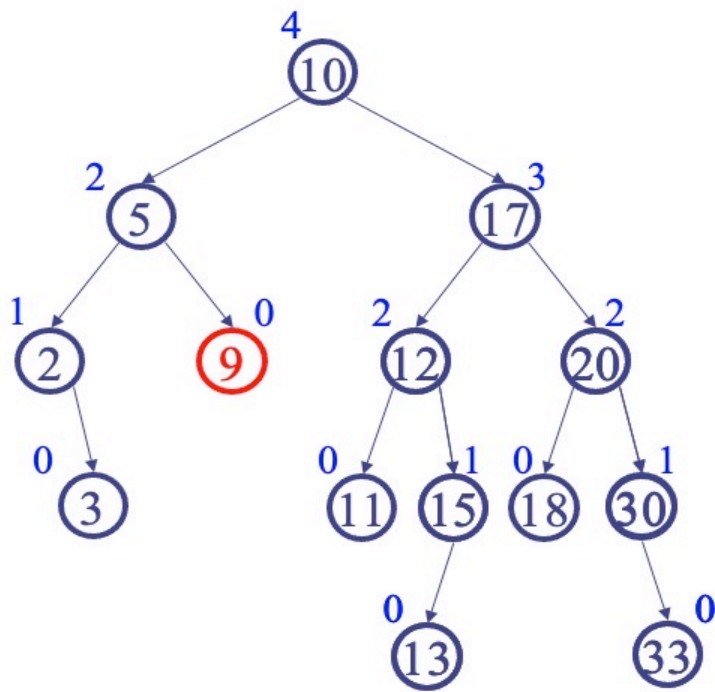
- ▶ Case 1: single rotation (Delete **12**)





# Deletion on AVL-tree

## ► Case 2: double rotation (Delete 9)

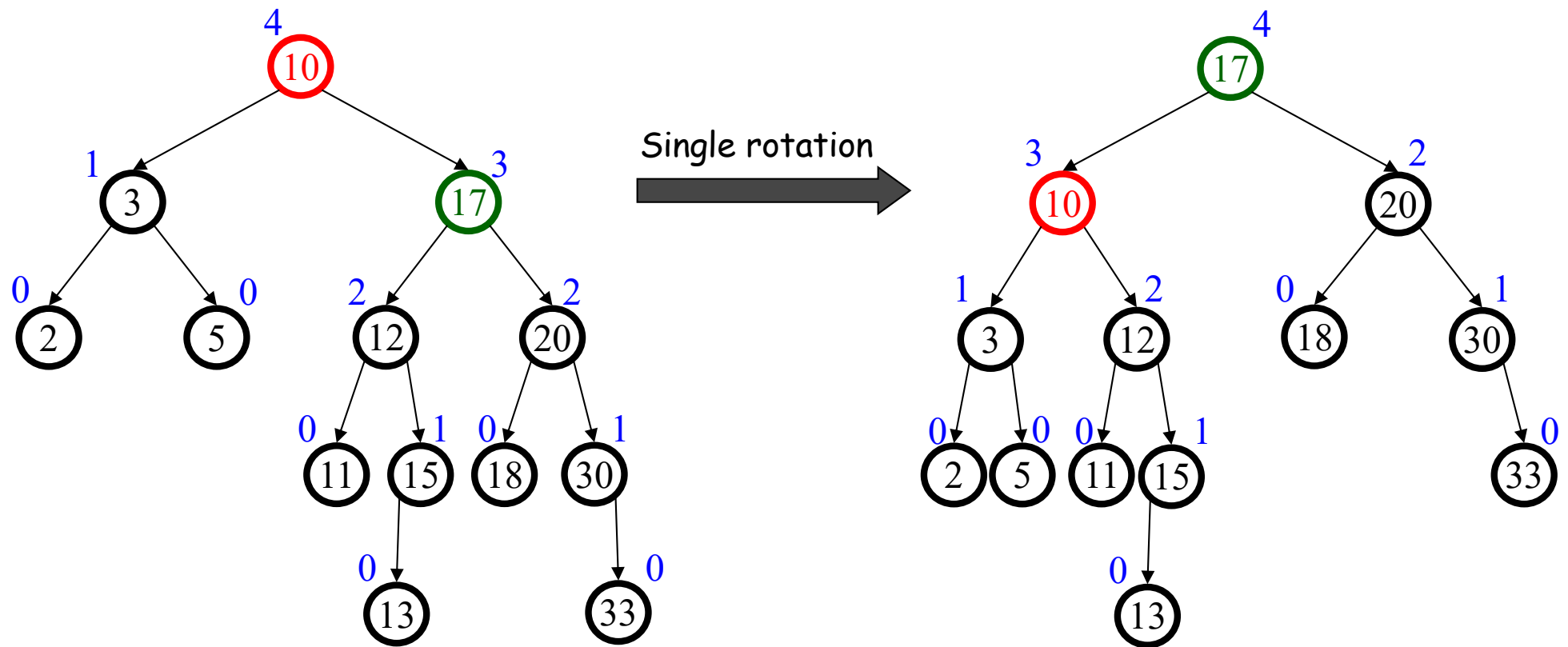


We get to choose whether to single or double rotate!



# Deletion on AVL-tree

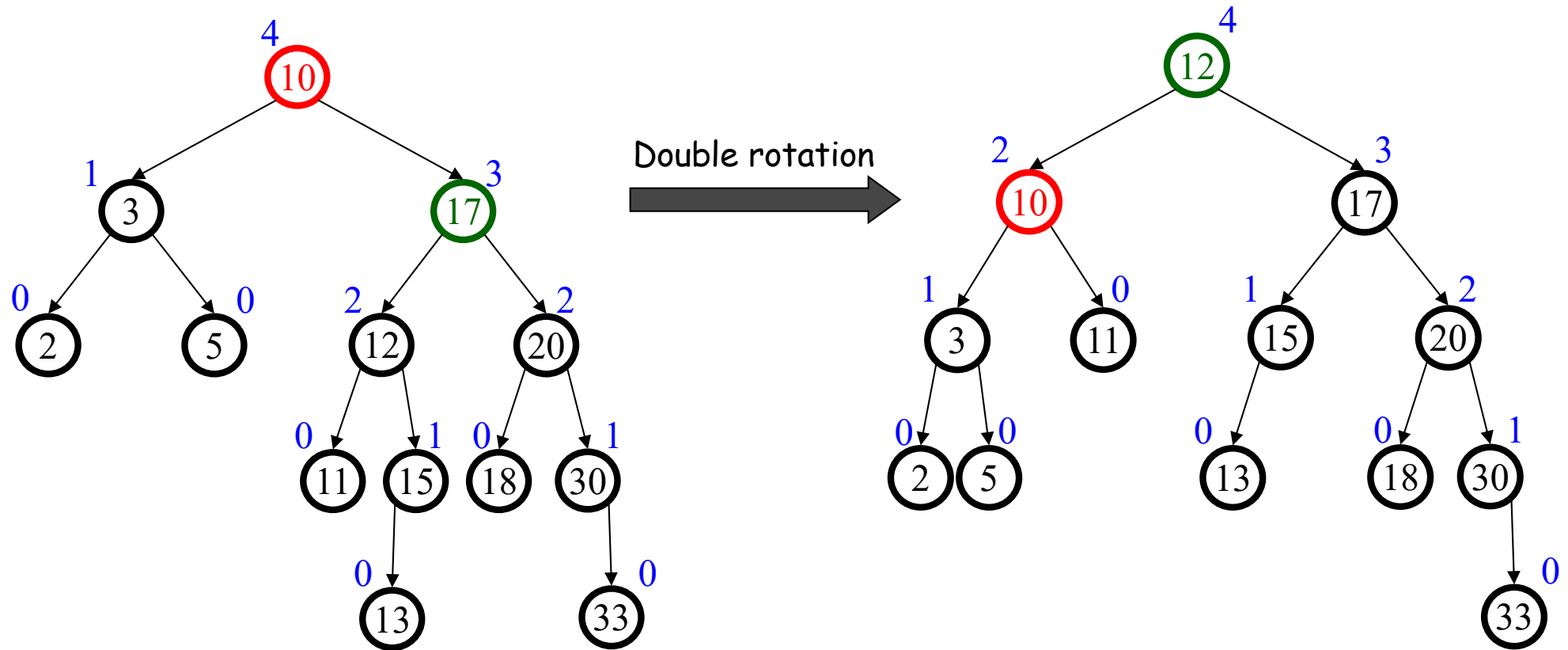
## ► Propagated single rotation





# Deletion on AVL-tree

## ► Propagated double rotation





# Deletion on AVL-tree

---

## ▶ General steps

- Search downward for the tree
- Delete node (may replace it by its successor)
- Unwind, correcting heights as we go
  - If imbalance #1,
    - Single rotation
  - If imbalance #2
    - Double rotation

## ▶ Homework:

- Implement insertion/deletion algorithms on AVL-tree
- Analyze the time complexity step by step



# Pros and cons of AVL trees

---

## ► Arguments for AVL trees:

1. Search is  $O(\log n)$  since AVL trees are always balanced
2. Insertion and deletions also cost  $O(\log n)$  time
3. The height balancing adds no more than a constant factor to the speed of insertion

## ► Arguments against using AVL trees:

1. Difficult to program & debug; more space for balance factor
2. Asymptotically faster but rebalancing costs time
3. Most large searches are done in database systems on disk and use other structures (e.g., B-trees)



# Recommended reading

---

- ▶ Reading this week
  - Chapter 12, textbook
- ▶ Next lecture
  - Red-black tree: chapter 13, textbook