CMSC 420 – 0201 – Fall 2019
Lecture 05

AVL Trees

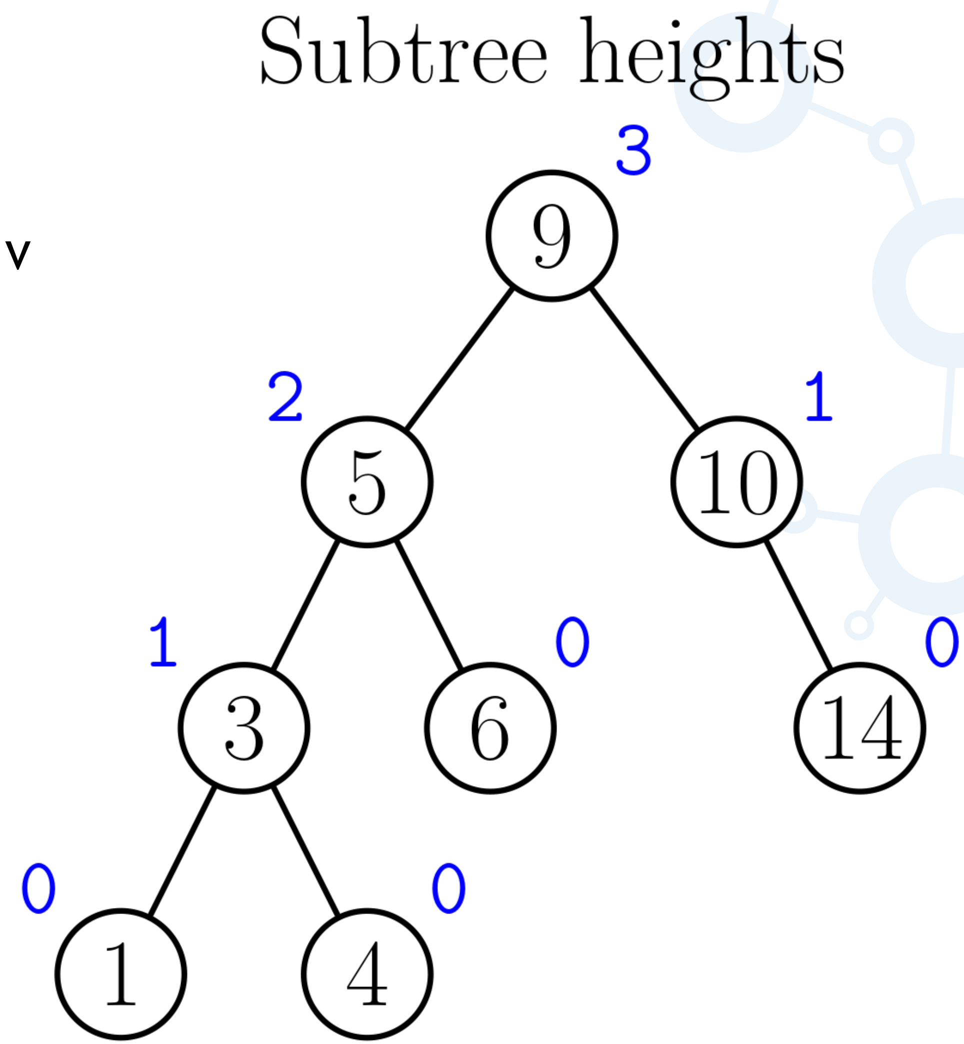
Balanced Binary Search Trees

- Standard binary search trees provide a simple implementation of the sorted dictionary abstract data type, but their worst-case performance is poor, $O(n)$ per operation
- Is there a version of the binary search tree that achieves $O(\log n)$ worst-case performance for all operations?
- Yes! Today we will study the oldest of these, AVL Trees, by Adelson-Velskii and Landis (1962)

AVL Tree

Height-Balanced Binary Search Tree

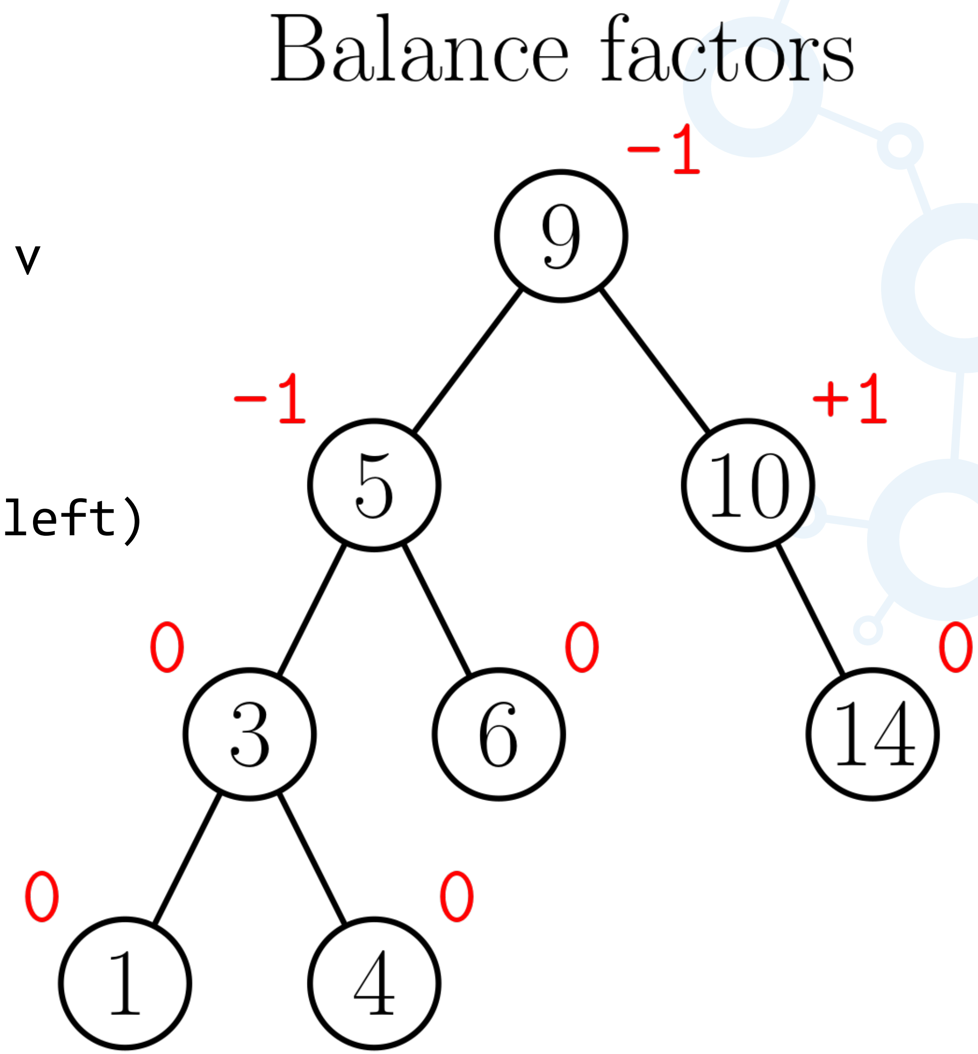
- Height:
  - `height(v)` is the height of the subtree rooted at v
  - `height(null)` = −1



AVL Tree

Height-Balanced Binary Search Tree

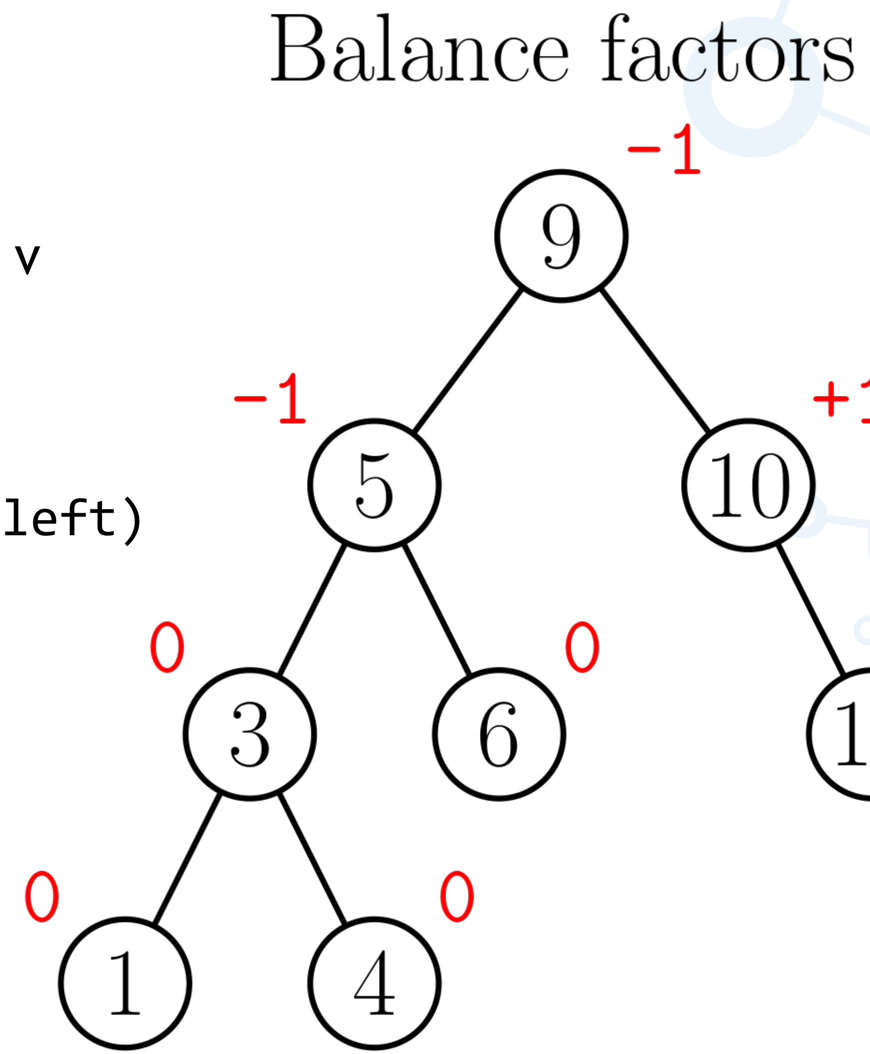
- Height:
  - height(v) is the height of the subtree rooted at v
  - height(null) = −1
- Balance Factor:
  - balance(v) = height(v.right) - height(v.left)
  - balance(v) < 0: Left-heavy
  - balance(v) > 0: Right-heavy



AVL Tree

Height-Balanced Binary Search Tree

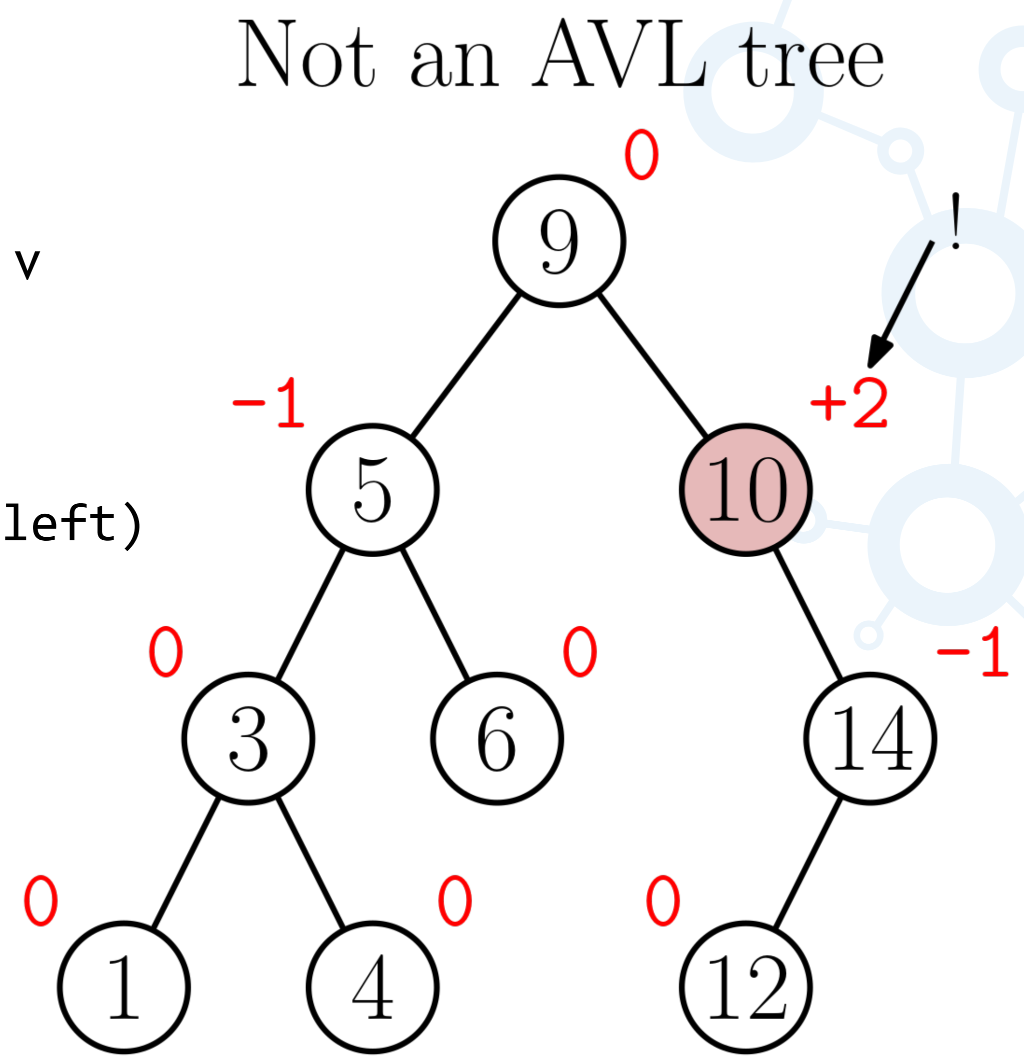
- Height:
  - `height(v)` is the height of the subtree rooted at v
  - `height(null)` $= -1$
- Balance Factor:
  - `balance(v) = height(v.right) - height(v.left)`
- AVL Height Condition:
  - For all nodes v, $-1 \leq$ `balance(v)` $\leq +1$



AVL Tree

Height-Balanced Binary Search Tree

- Height:
  - `height(v)` is the height of the subtree rooted at v
  - `height(null)` $= -1$
- Balance Factor:
  - `balance(v) = height(v.right) – height(v.left)`
- AVL Height Condition:
  - For all nodes v, $-1 \leq$ `balance(v)` $\leq +1$

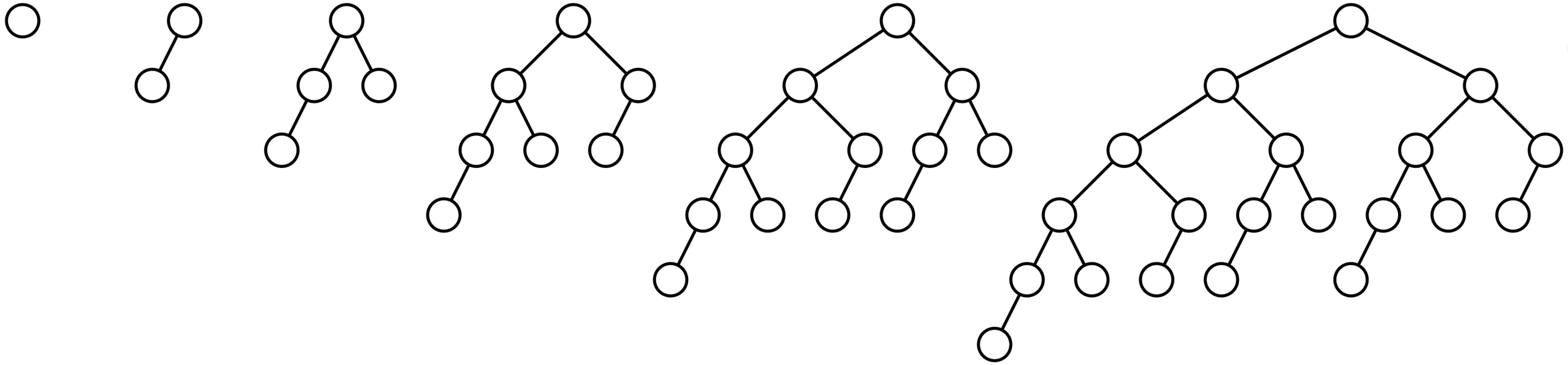


Worst-Case Height

Does height condition imply O(log n) height?

- Consider the AVL trees of height h with the fewest possible nodes:

Height:	0	1	2	3	4	5
# Nodes:	1	2	4	7	12	20
# Leaves	1	1	2	3	5	8

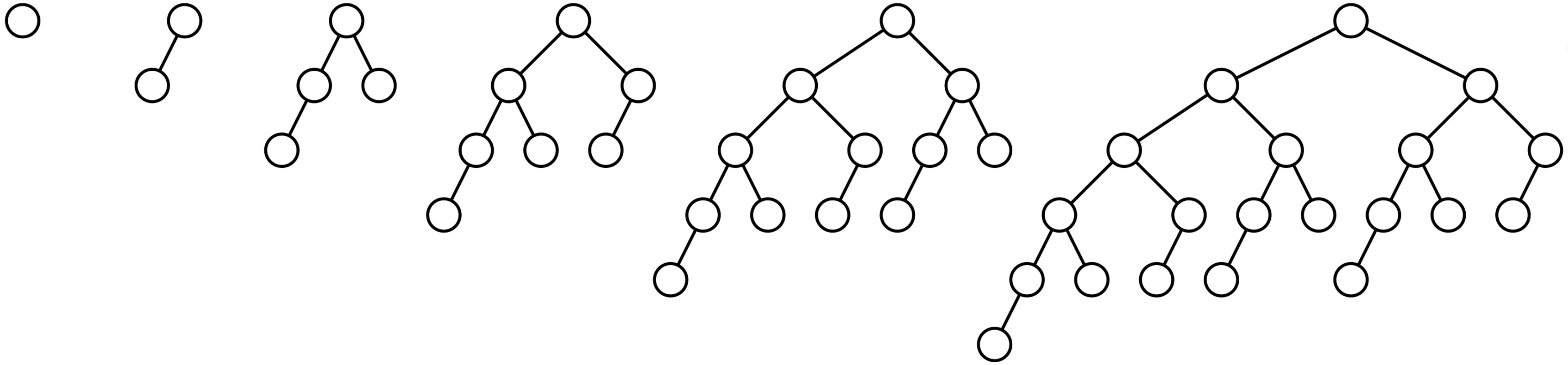


Worst-Case Height

Does height condition imply O(log n) height?

- Consider the AVL trees of height h with the fewest possible nodes:

Height:	0	1	2	3	4	5
# Nodes:	1	2	4	7	12	20
# Leaves	1	1	2	3	5	8



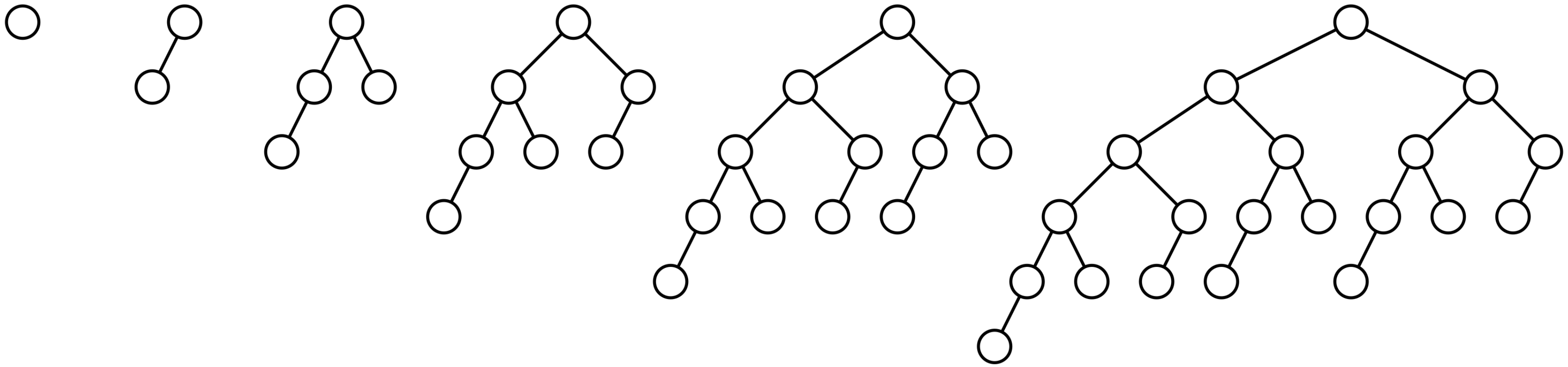
- $N(h)$ = minimum number of nodes for tree of height h
- Lemma: $N(h) \approx \varphi^h$, where $\varphi = \frac{1+\sqrt{5}}{2}$ (Golden Ratio!)

Worst-Case Height

Does height condition imply O(log n) height?

- **Lemma**: $N(h) \approx \varphi^h$, where $\varphi = \frac{1+\sqrt{5}}{2}$
- **Theorem**: Maximum height of a tree with n nodes is $O(\log n)$

Height:	0	1	2	3	4	5
# Nodes:	1	2	4	7	12	20
# Leaves	1	1	2	3	5	8

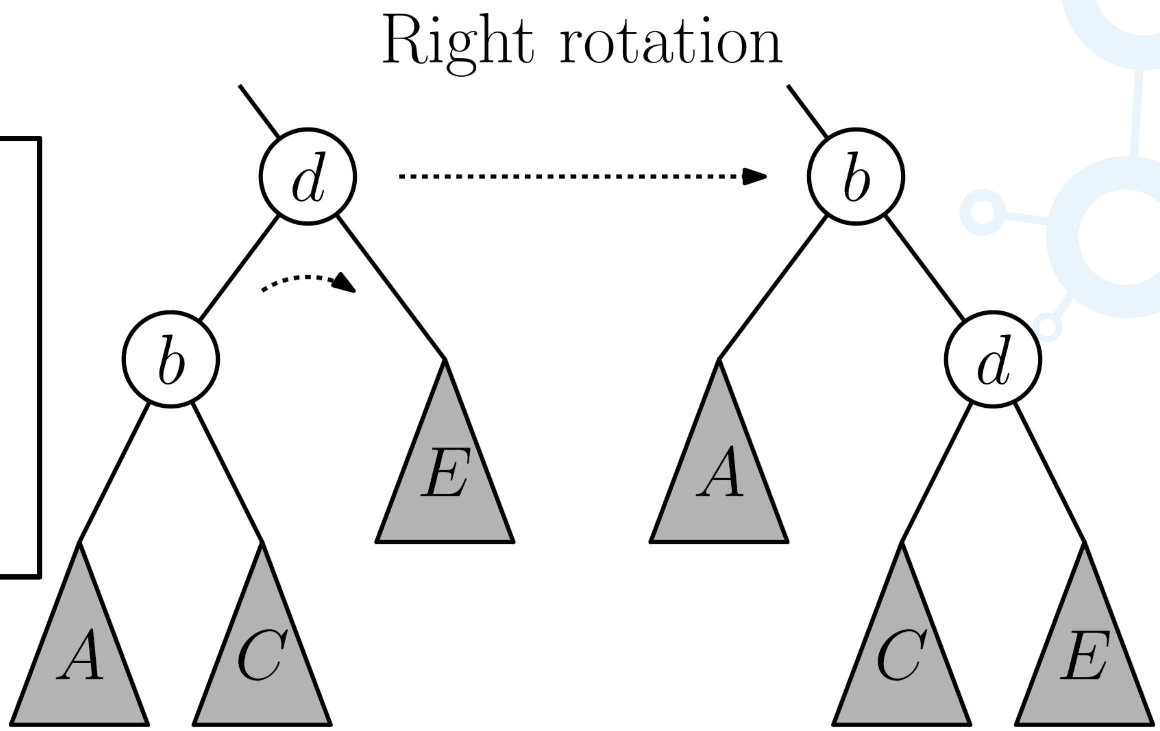


Rotation

Single Rotation

- When tree is out of balance, we need an operation that modifies subtree heights while preserving tree's inorder properties
- Rotation: (Also called single rotation)

```
AvlNode rotateRight(AvlNode p)
{
        AvlNode q = p.left;
        p.left = q.right;
        q.right = p;
        return q;
}
```

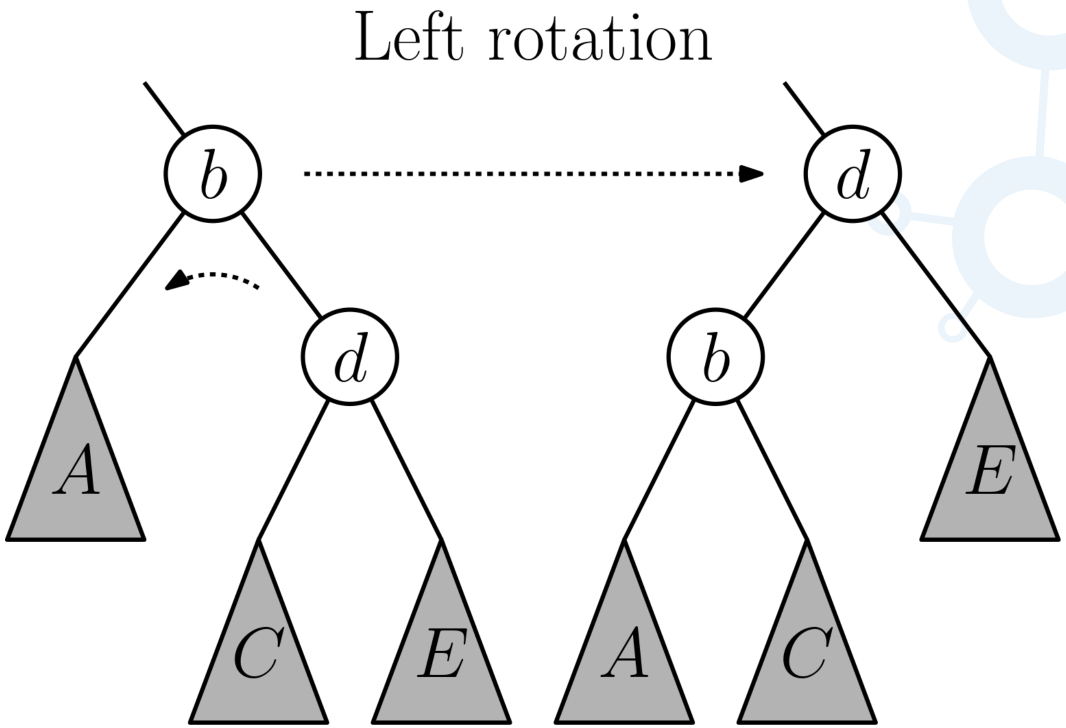


Rotation

Single Rotation

- When tree is out of balance, we need an operation that modifies subtree heights while preserving tree's inorder properties
- Rotation: (Also called single rotation)

```
AvlNode rotateLeft(AvlNode p)
{
        AvlNode q = p.right;
        p.right = q.left;
        q.left = p;
        return q;
}
```

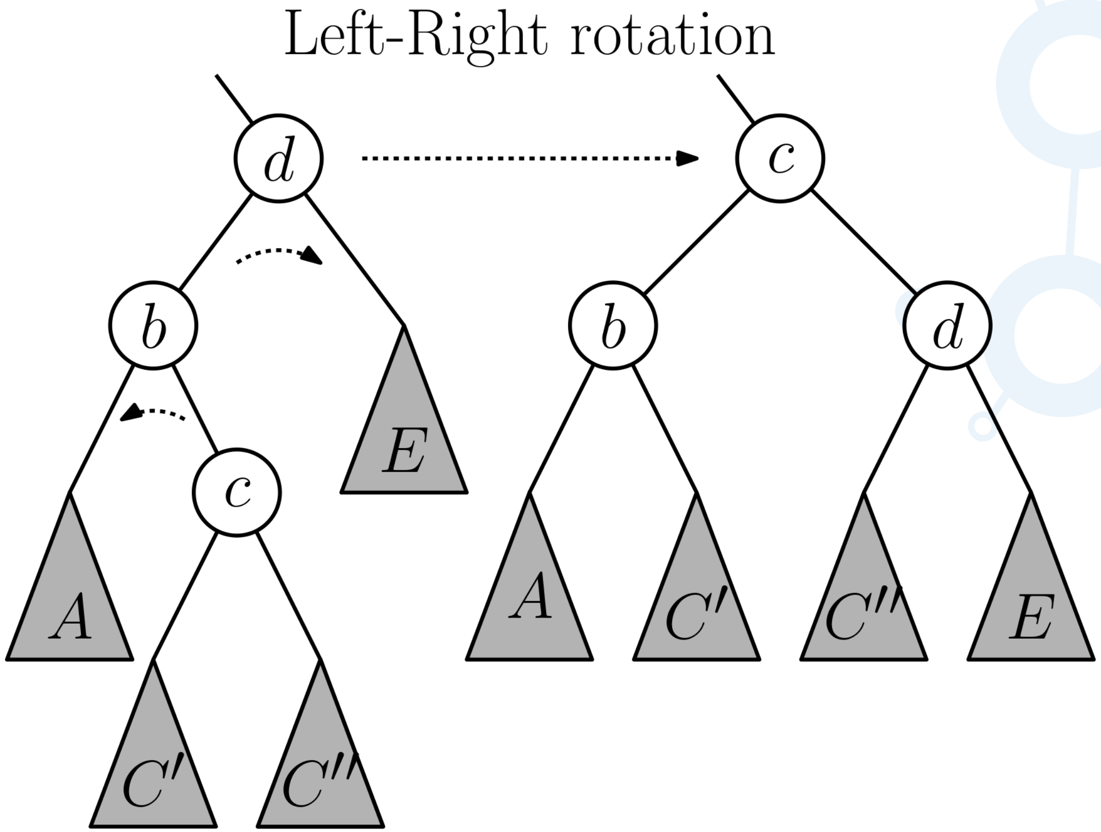


Rotation

Double Rotation

- A single rotation does not always suffice
- Double Rotation:

```
AvlNode rotateLeftRight(AvlNode p)
{
  p.left = rotateLeft(p.left);
  return rotateRight(p);
}
```

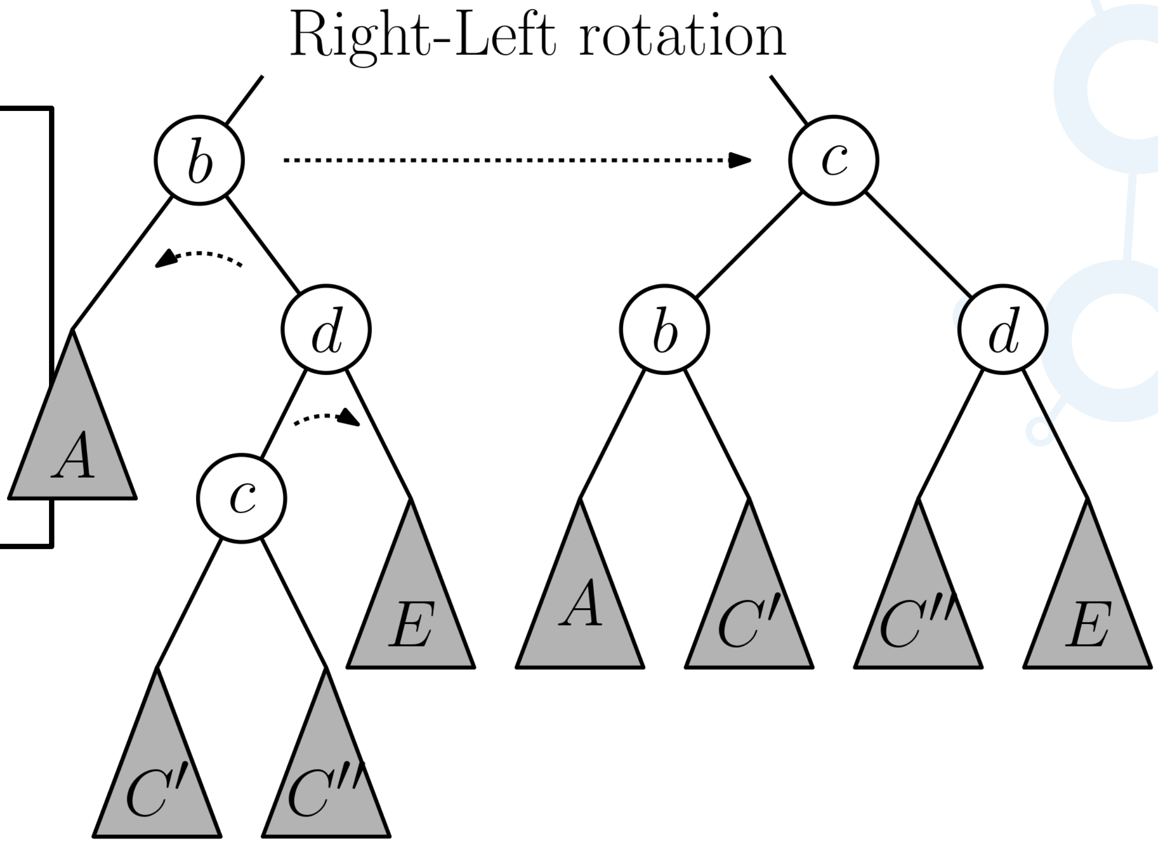


Rotation

Double Rotation

- A single rotation does not always suffice
- Double Rotation:

```
AvlNode rotateRightLeft(AvlNode p)
{
  p.right = rotateRight(p.right);
  return rotateLeft(p);
}
```

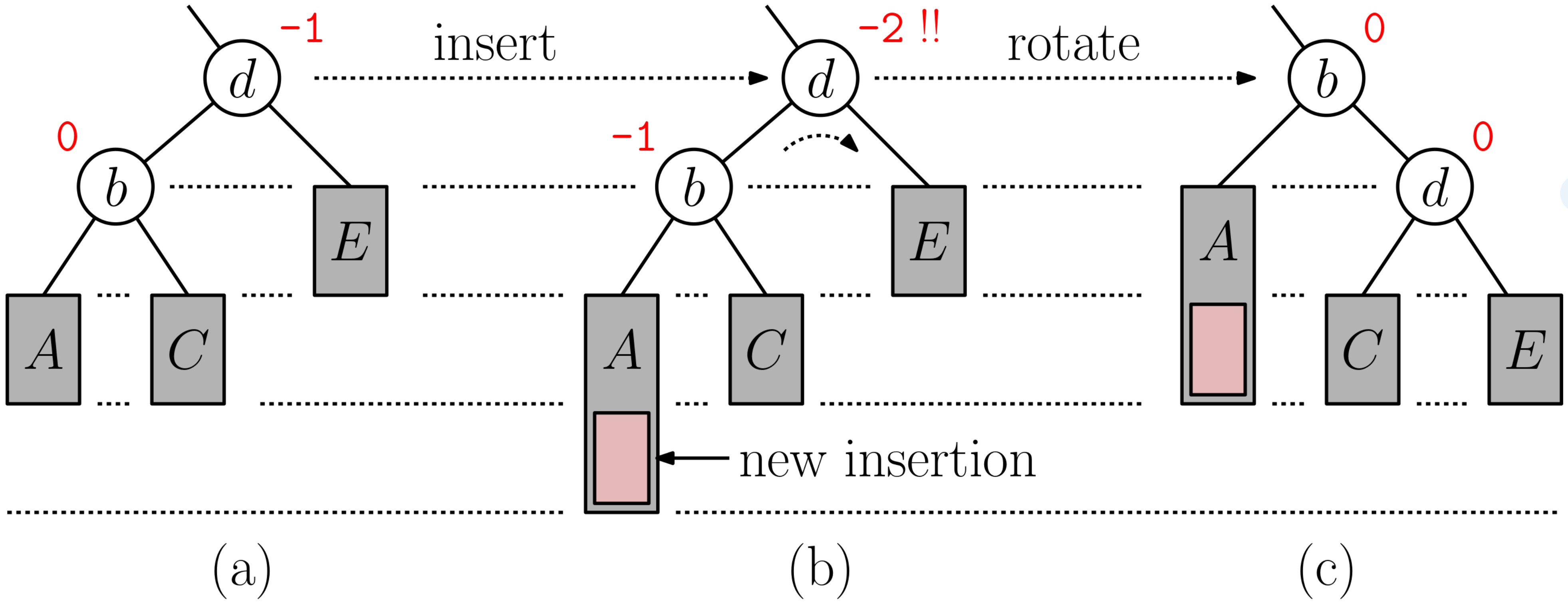


AVL Insertion

- Insert the key-value pair as in standard binary search trees
- As we return, update heights/balance factors along the search path
- If any node is found to be out of balance (balance factor $= -2\ or\ +2$), apply appropriate rotations to restore balance
- Cases: (Messy, but a lot of symmetry)
  - Left heavy (`balance(p) == −2`)
    - Left-left grandchild as heavy as left-right grandchild
    - Left-left grandchild lighter than left-right grandchild
  - Right heavy (`balance(p) == +2`)
    - Right-right grandchild as heavy as right-left grandchild
    - Right-right grandchild lighter than right-left grandchild

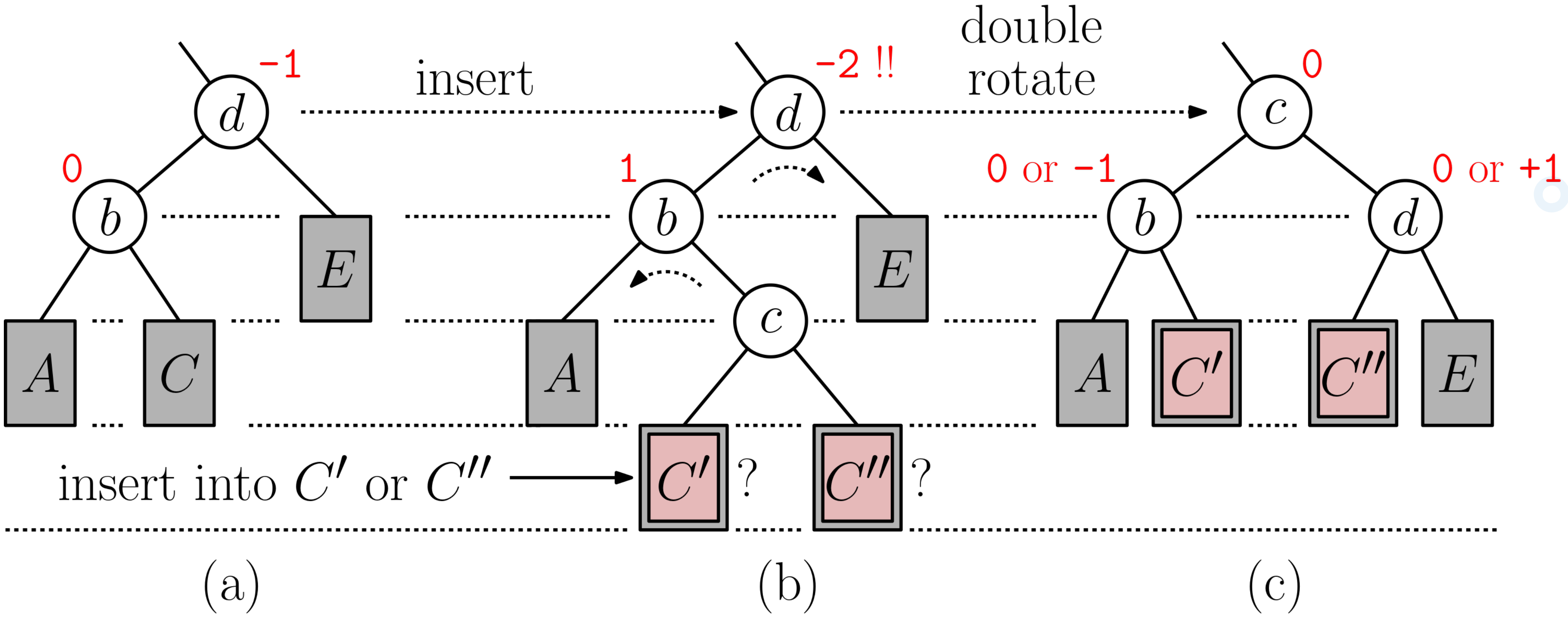
AVL Insertion

- Left-left (A) as heavy as left-right (C) - Fix with single rotation



AVL Insertion

- Left-left (A) lighter than left-right (C) - Fix with double rotation



AVL Insertion

Utility functions

```
int height(AvlNode p) {                         // height of subtree (-1 if null)
    return p == null ? -1 : p.height;
}

void updateHeight(AvlNode p) {                  // update height from children
    p.height = 1 + max(height(p.left), height(p.right));
}

int balanceFactor(AvlNode p) {                  // compute balance factor
    return height(p.right) - height(p.left);
}
```

AVL Insertion

Insertion function

```
AvlNode insert(Key x, Value v, AvlNode p) {
        if (p == null) {                    // fell out of tree; create node
            p = new AvlNode(x, v, null, null);
        }
        else if (x < p.key) {               // x is smaller - insert left
            p.left = insert(x, p.left);     // ... insert left
        else if (x > p.key) {               // x is larger - insert right
            p.right = insert(x, p.right);   // ... insert right
        }
        else throw DuplicateKeyException;   // key already in the tree?
        return rebalance(p);                // rebalance if needed
}
```

AVL Insertion

Rebalancing function

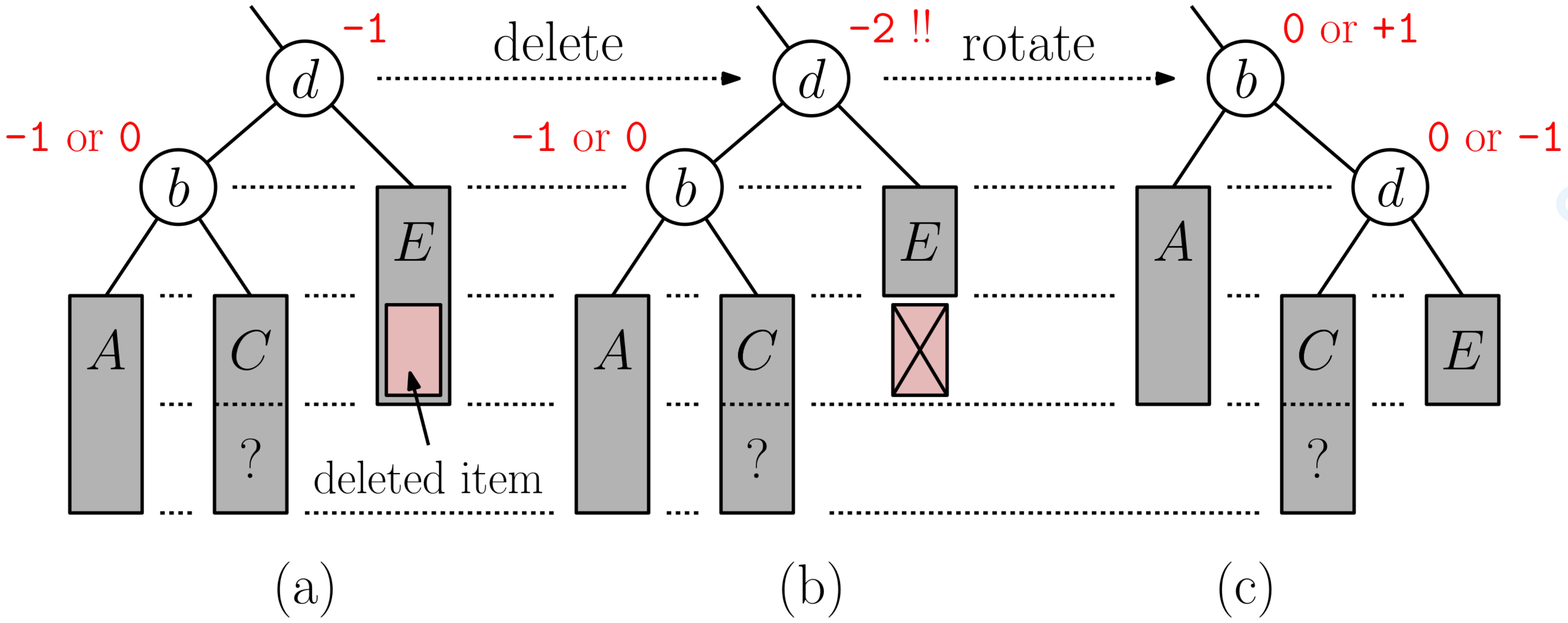
```
AvlNode rebalance(AvlNode p) {
    if (p == null) return p;                  // null - nothing to do
    if (balanceFactor(p) < -1) {              // too heavy on the left?
        if (height(p.left.left) >= height(p.left.right)) { // left-left heavy?
            p = rotateRight(p);               // fix with single rotation
        else                                  // left-right heavy?
            p = rotateLeftRight(p);           // fix with double rotation
    } else if (balanceFactor(p) > +1) {       // too heavy on the right?
        if (height(p.right.right) >= height(p.right.left)) { // right-right?
            p = rotateLeft(p);                // fix with single rotation
        else                                  // right-left heavy?
            p = rotateRightLeft(p);           // fix with double rotation
    }
    updateHeight(p);                          // update p's height
    return p;
}
```

AVL Deletion

- Delete the key-value pair as in standard binary search trees
- As we return, update heights/balance factors along the search path
- If any node is found to be out of balance (balance factor $= -2\ or\ +2$), apply appropriate rotations to restore balance
- Cases: (Same as for insertion)
  - Left heavy (`balance(p) == −2`)
    - Left-left grandchild as heavy as left-right grandchild
    - Left-left grandchild lighter than left-right grandchild
  - Right heavy (`balance(p) == +2`)
    - Right-right grandchild as heavy as right-left grandchild
    - Right-right grandchild lighter than right-left grandchild

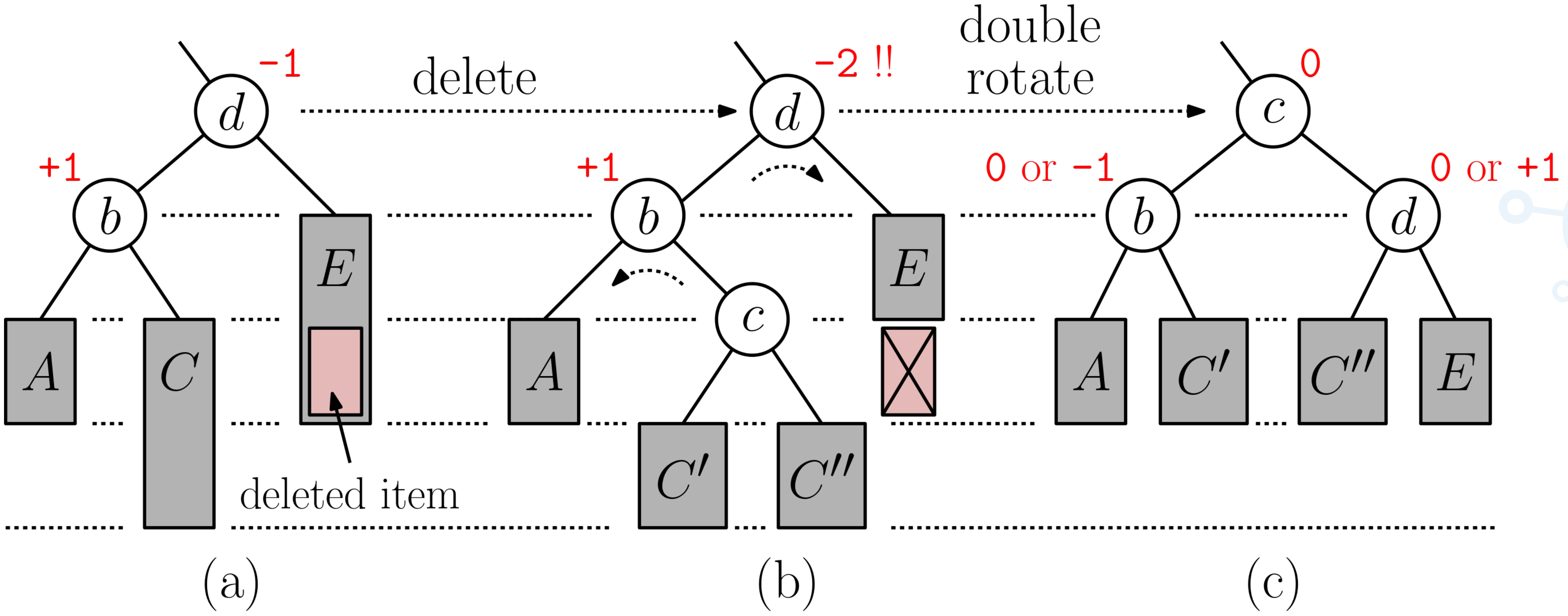
AVL Deletion

- Left-left (A) as heavy as left-right (C) - Fix with single rotation



AVL Deletion

- Left-left (A) lighter than left-right (C) - Fix with double rotation



Summary

- AVL Tree definition – Balance condition
- AVL trees have O(log n) height
- Rotations
  - Single rotation
  - Double rotations
- AVL insertion
- AVL deletion