Middlebury

CSCI 201: Data Structures

Spring 2025

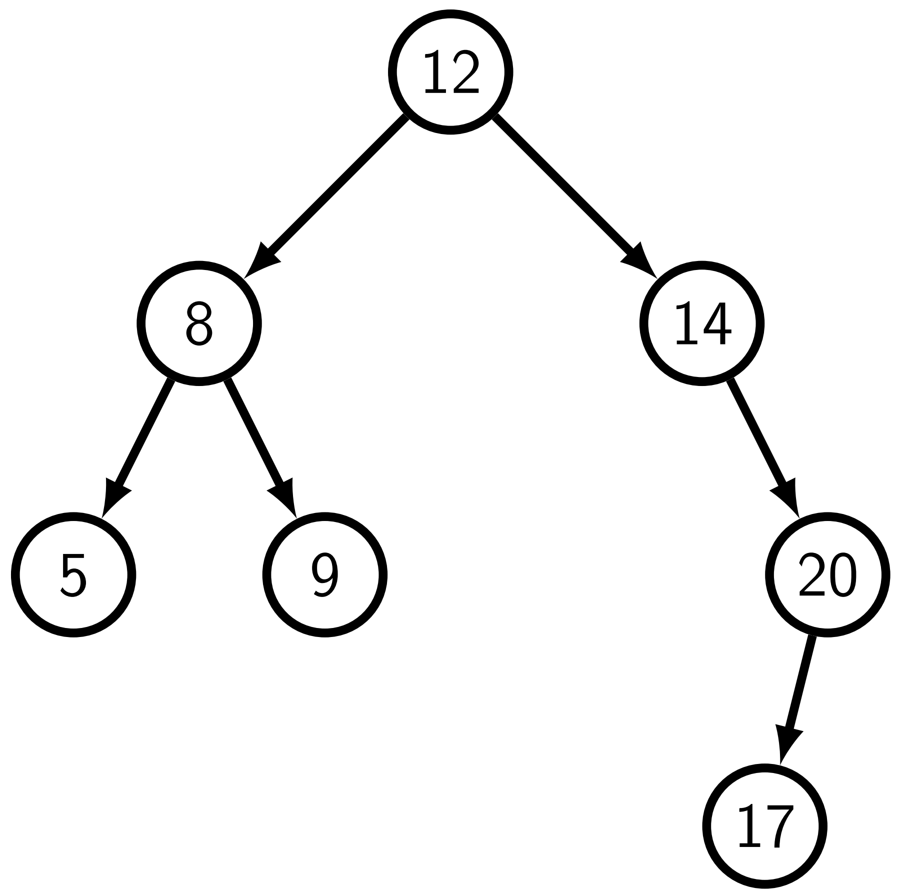
Lecture 9W: Balanced Trees

Goals for today:

- Revisit `remove` method, and how to implement?
- Analyze the complexity of `contains`, `add`, `remove` in a BST.
- **Motivation for balancing:** what happens if we insert keys in a BST in a certain way?
- Calculate and save the height of a node when it's added.
- Calculate the *balance factor* of a node.
- Perform **rotations** on a tree to improve the balance factor.
- Implement a Adelson-Velsky-Landis (AVL) self-balancing binary search tree.

Removing an item/key from a BST:

- Start at `node = root`.
- If `key < node.key`, need to remove node in left subtree (`node.left`).
- If `key > node.key`, need to remove node in right subtree (`node.right`).
- If `key == node.key`, three cases to consider:
    1. Is this a leaf? Delete node.
    2. (a) If `node.left == null`, "graft" `node.right` to the `node`.
       (b) If `node.right == null`, "graft" `node.left` to the `node`.
    3. Two (non-`null`) child nodes?
        - Find node (`minNode`) with minimum `key` in `node.right` (right subtree).
        - Replace `node.key` with `minNode.key`.
        - Now we remove `minNode` from right subtree.



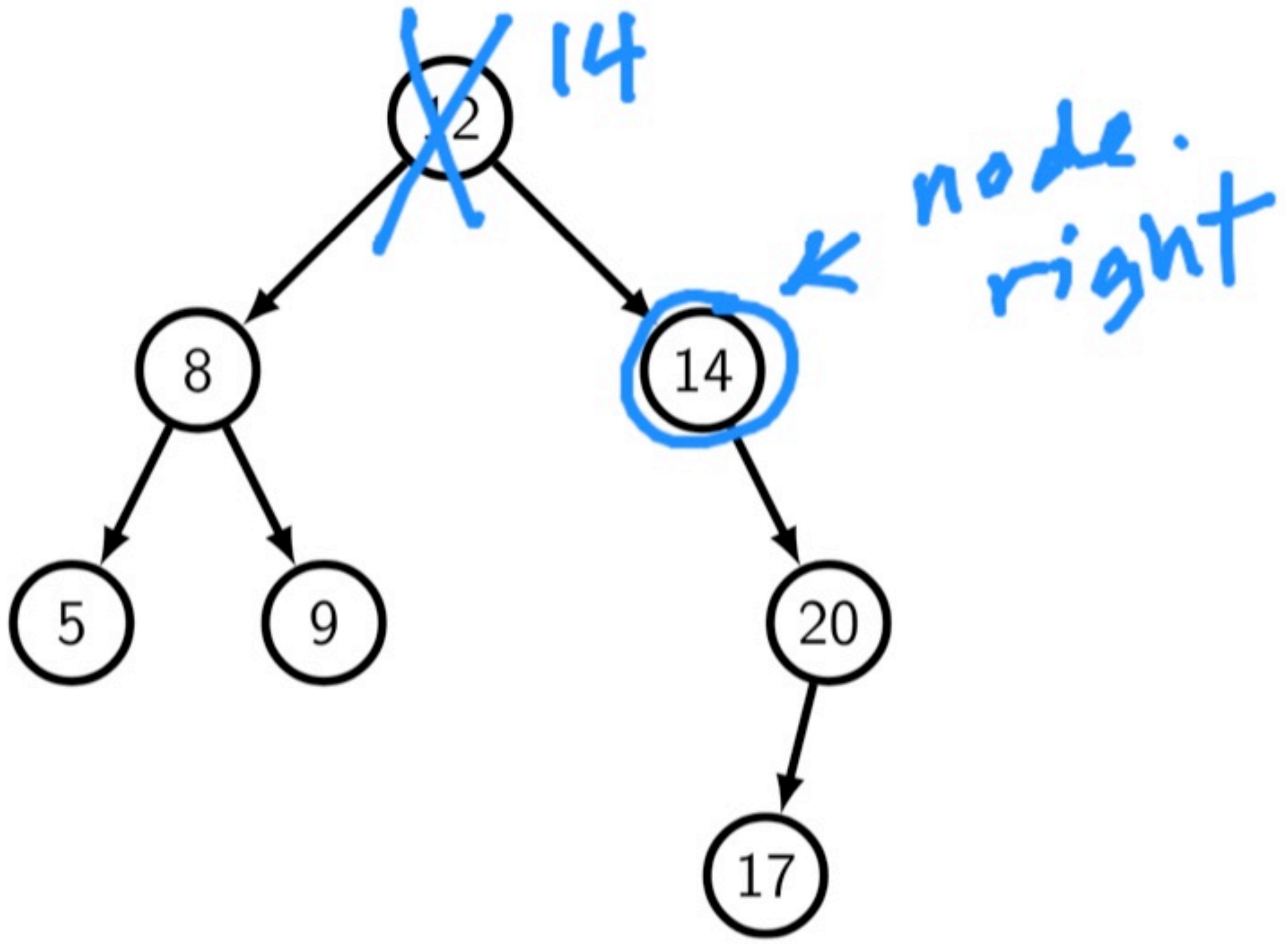
Implementing the remove method.

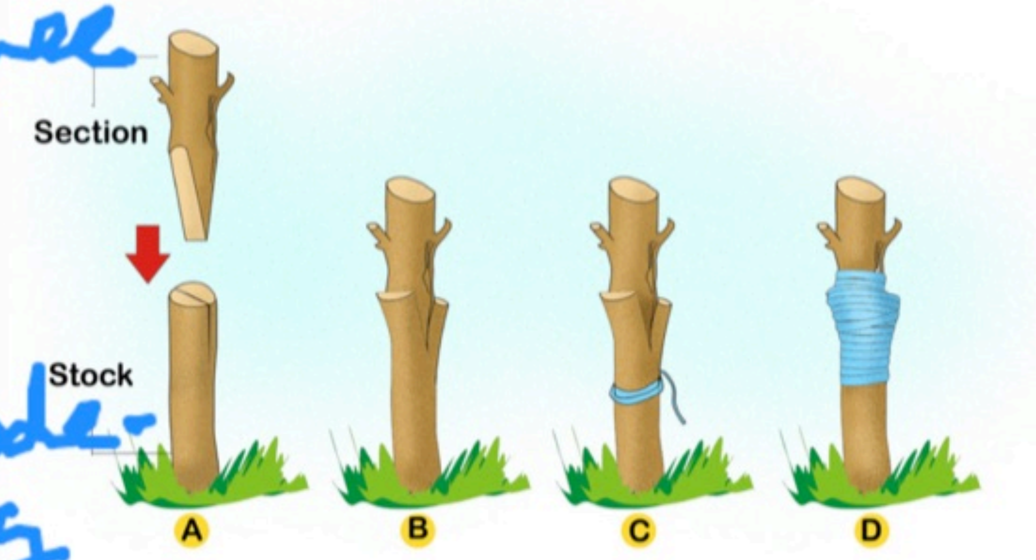
```java
public void remove(E key) {
  root = remove(root, key);
}

private Node remove(Node node, E key) {

  if (node == null) {
    return null; // went beyond a leaf, key not found
  }

  int compareResult = key.compareTo(node.key);
  if (compareResult < 0) {
    // node to remove is (possibly) in the left subtree
    node.left = remove(node.left, key);
  } else if (compareResult > 0) {
    // node to remove is (possibly) in the right subtree
    node.right = remove(node.right, key);
  } else {
    // we are at the node to delete, check 3 cases
    if (node.left == null && node.right == null) {
      return null; // case 1, delete a leaf
    } else if (node.left == null) {
      return node.right; // case 2a
    } else if (node.right == null) {
      return node.left; // case 2b
    } else {

      // case 3: what goes here?

    }
  }
  return node;
}
```

a. find minNode in right subtree

b. set node.key = minNode.key

c. call remove on right subtree and save into node.right



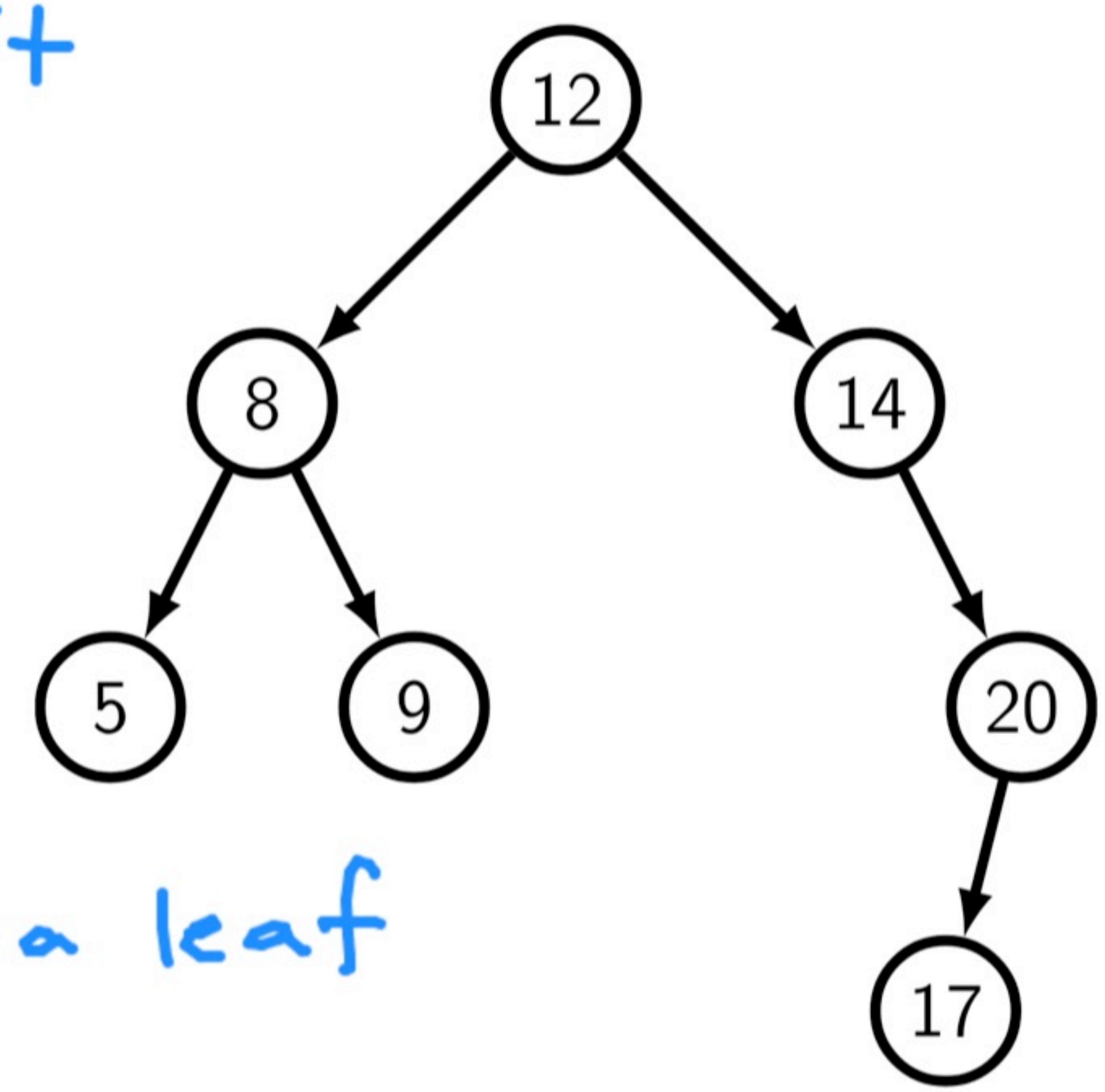


Last time we talked about **`contains`**, **`add`**, **`remove`**.
What is the runtime complexity?

<u>contains</u> : search left or right
O(height)

<u>add</u> : adds a leaf
O(height)

<u>remove</u> : can go to a leaf
O(height)



height can be n or log(n)

→ O(log n) if balanced

Which of the following `contains` methods will *work* for these types of trees.

if tree is balanced
O(logn)

Method 1:

```java
public boolean contains(E key) {
  return contains(root, key);
}

private boolean contains(Node node, E key) {
  if (node == null) return false;

  int compareResult = key.compareTo(node.key);
  if (compareResult == 0) {
    return true;
  } else if (compareResult < 0) {
    return contains(node.left, key);
  } else {
    return contains(node.right, key);
  }
}
```

Method 2:

```java
public boolean contains(E key) {
  return contains(root, key);
}

private boolean contains(Node node, E key) {
  if (node == null) return false;

  if (key.equals(node.key)) return true;

  return contains(node.left, key) ||
         contains(node.right, key);
}
```

O(n)
might check every node

- **General** binary tree: no specific properties. Method 2
- **Heap** (min or max): complete, every node value > child node values (for max-heap). Method 2
- **Binary search tree**: left subtree keys < right subtree keys of every node. Methods 1, 2

Motivation for balancing: what happens if we do this?

```java
int n = 20;
for (int i = 0; i < n; i++) {
  tree.add(i);
}
System.out.println(tree);
```

```
└──(0)
|   └──(1)
|   |   └──(2)
|   |   |   └──(3)
|   |   |   |   └──(4)
|   |   |   |   |   └──(5)
|   |   |   |   |   |   └──(6)
|   |   |   |   |   |   |   └──(7)
|   |   |   |   |   |   |   |   └──(8)
|   |   |   |   |   |   |   |   |   └──(9)
|   |   |   |   |   |   |   |   |   |   └──(10)
|   |   |   |   |   |   |   |   |   |   |   └──(11)
|   |   |   |   |   |   |   |   |   |   |   |   └──(12)
|   |   |   |   |   |   |   |   |   |   |   |   |   └──(13)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   └──(14)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   └──(15)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   └──(16)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   └──(17)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   └──(18)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   └──(19)
```

0 \ 1 \ 2 \ 3 \ 4 \ ...

Ideally, we'd like to maintain a *balanced* tree to reduce the tree height.

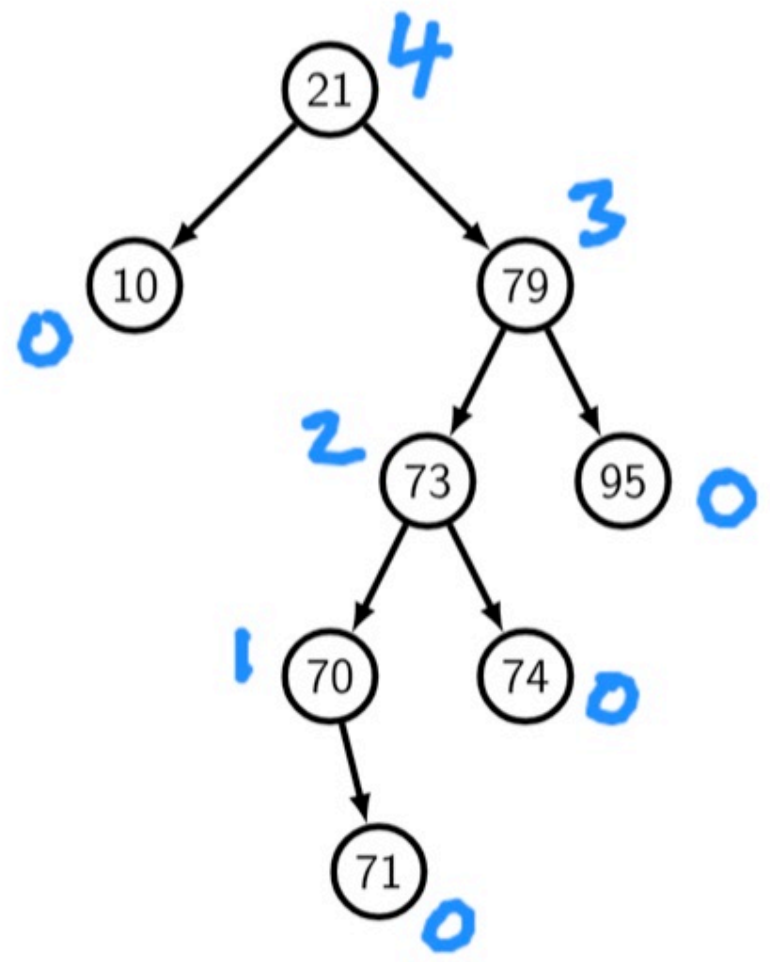
- Recall the **height** of a node (and tree): length of *longest* path from node to a leaf.
- **Balanced** means the height of the left and right subtrees differ by at most one.
- **Balance factor** (bf) of a node: `height(node.left) - height(node.right)`.
- height of a `null` node is `-1`.
- `bf < 0`: right-heavy, `bf > 0`: left-heavy.

height of null
-1
height of leaf
0
otherwise
1 + max (hLeft, hRight)

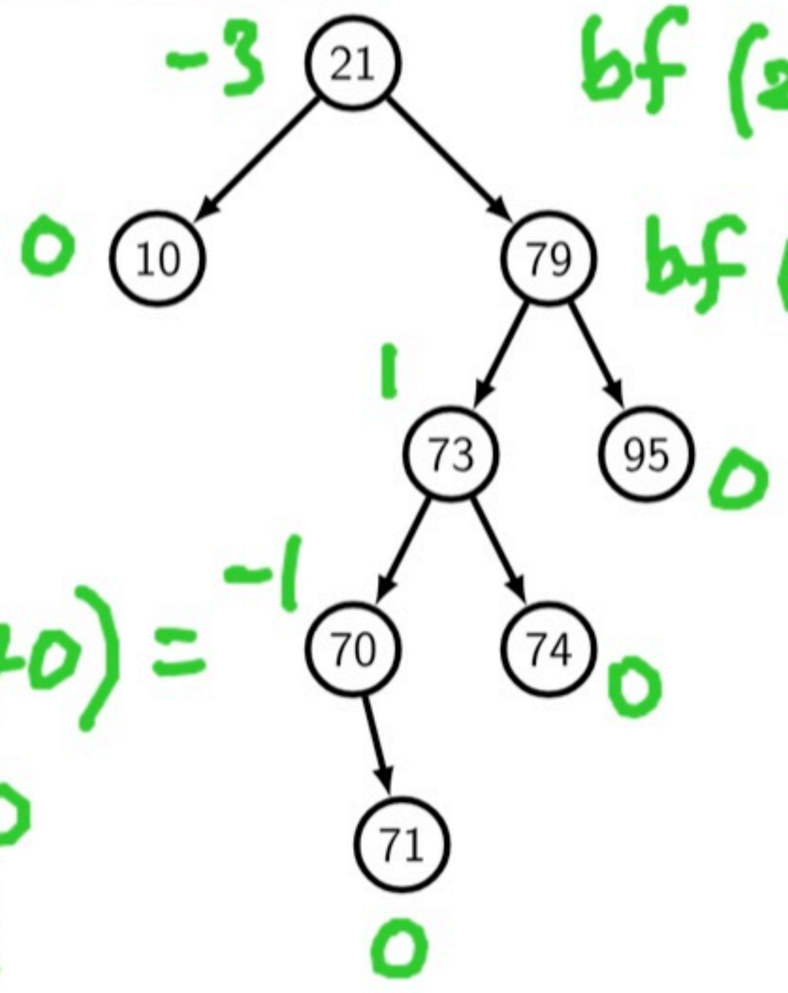
AVL tree

maintain a balance factor of -1, 0, +1 at all nodes

height



balance factor



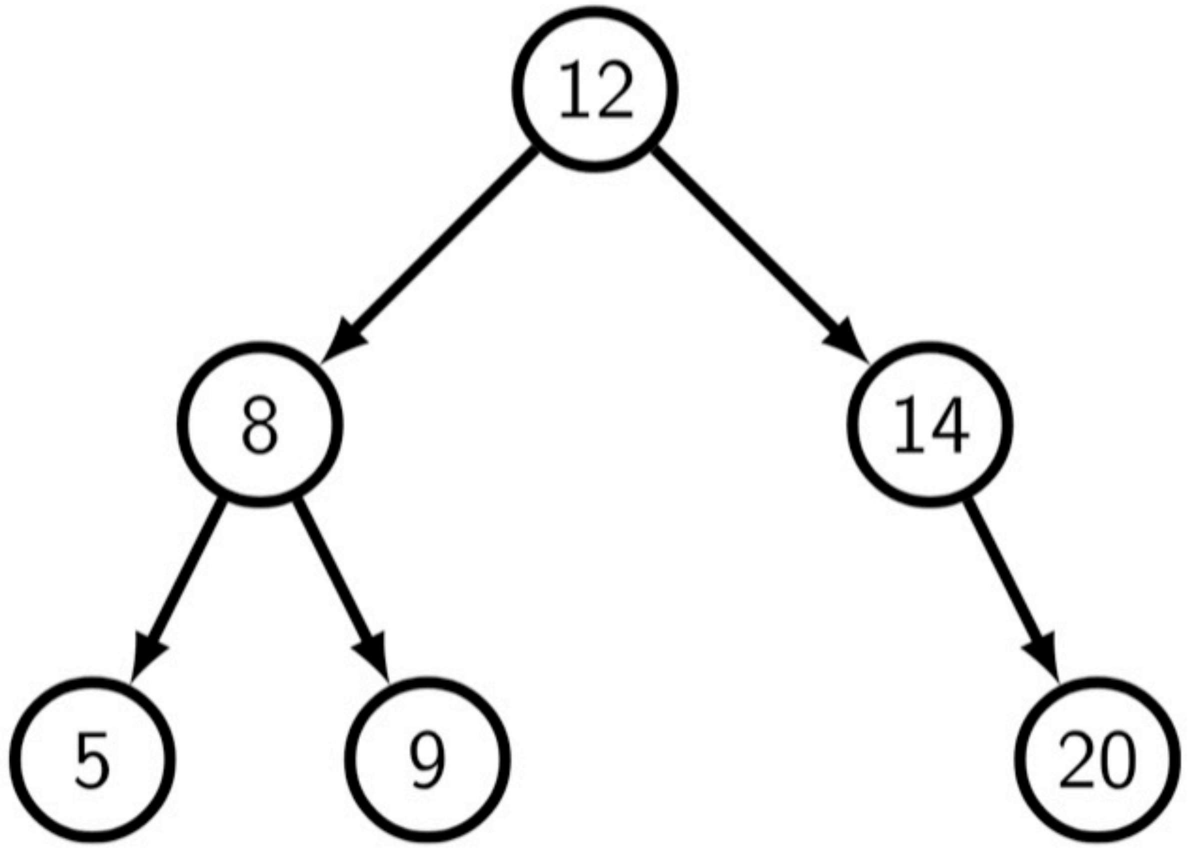
bf (21) = 0 - 3 = -3

bf (79) = 2 - 0 = 2

bf (70) =
-1 - 0
= -1

Back to our add method: let's keep track of height after updating a node (as we recurse back up the tree).

- Start at `node = root`.
- If `key < node.key`, need to put in left subtree.
- If `key > node.key`, need to put in right subtree.
- If `key == node.key`, key is already in tree! Done.
- If `node == null`, create a new `Node` for this `key`.
- Update the height of the node after insertion in left/right subtrees.




```java
class Node {
  public E key;
  public Node left;
  public Node right;
  public int height;  ← new field

  public Node(E key) {
    this.key = key;
  }

  public void calculateHeight() {
    // TODO
  }
}

private Node add(Node node, E key) {
  if (node == null) {
    return new Node(key);
  }

  int compareResult = key.compareTo(node.key);
  if (compareResult < 0) {
    node.left = add(node.left, key);
  } else if (compareResult > 0) {
    node.right = add(node.right, key);
  }

  // update the height of this node
  node.calculateHeight();

  return node;
}
```

Updating the height as we recurse back up the tree.

```java
class Node {
  public E key;
  public Node left;
  public Node right;
  public int height;

  public Node(E key) {
    this.key = key;
  }

  public void calculateHeight() {
    // TODO
  }
}

private Node add(Node node, E key) {
  if (node == null) {
    return new Node(key);
  }

  int compareResult = key.compareTo(node.key);
  if (compareResult < 0) {
    node.left = add(node.left, key);
  } else if (compareResult > 0) {
    node.right = add(node.right, key);
  }

  // update the height of this node
  node.calculateHeight();

  return node;
}
```

```java
public void calculateHeight() {
  if (left == null && right == null) {
    height = 0;
  } else {
    int hL = (left == null) ? 0 : node.left.calculateHeight();
    int hR = (right == null) ? 0 : node.right.calculateHeight();
    height = 1 + Math.max(hL, hR);
  }
}
```

```java
public void calculateHeight() {
  if (left == null && right == null) {
    height = 0;
  } else {
    int hL = (left == null) ? 0 : left.height;
    int hR = (right == null) ? 0 : right.height;
    height = 1 + Math.max(hL, hR);
  }
}
```

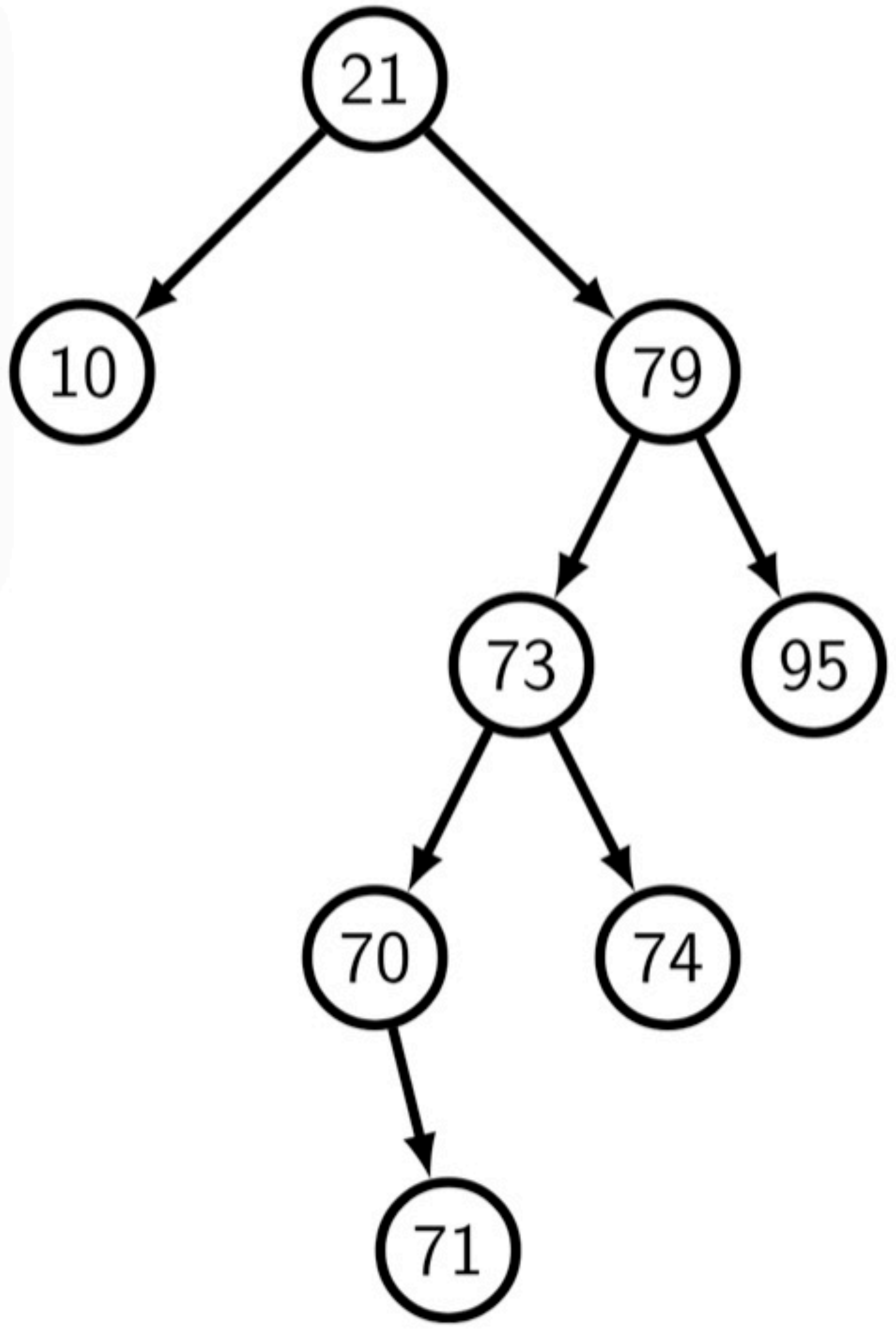
```java
public void calculateHeight() {
  int hL = (left == null) ? -1 : left.height;
  int hR = (right == null) ? -1 : right.height;
  height = 1 + Math.max(hL, hR);
}
```

Exercise: write a method called **getBalanceFactor** for the **Node** class.

BF = height of left − height of right

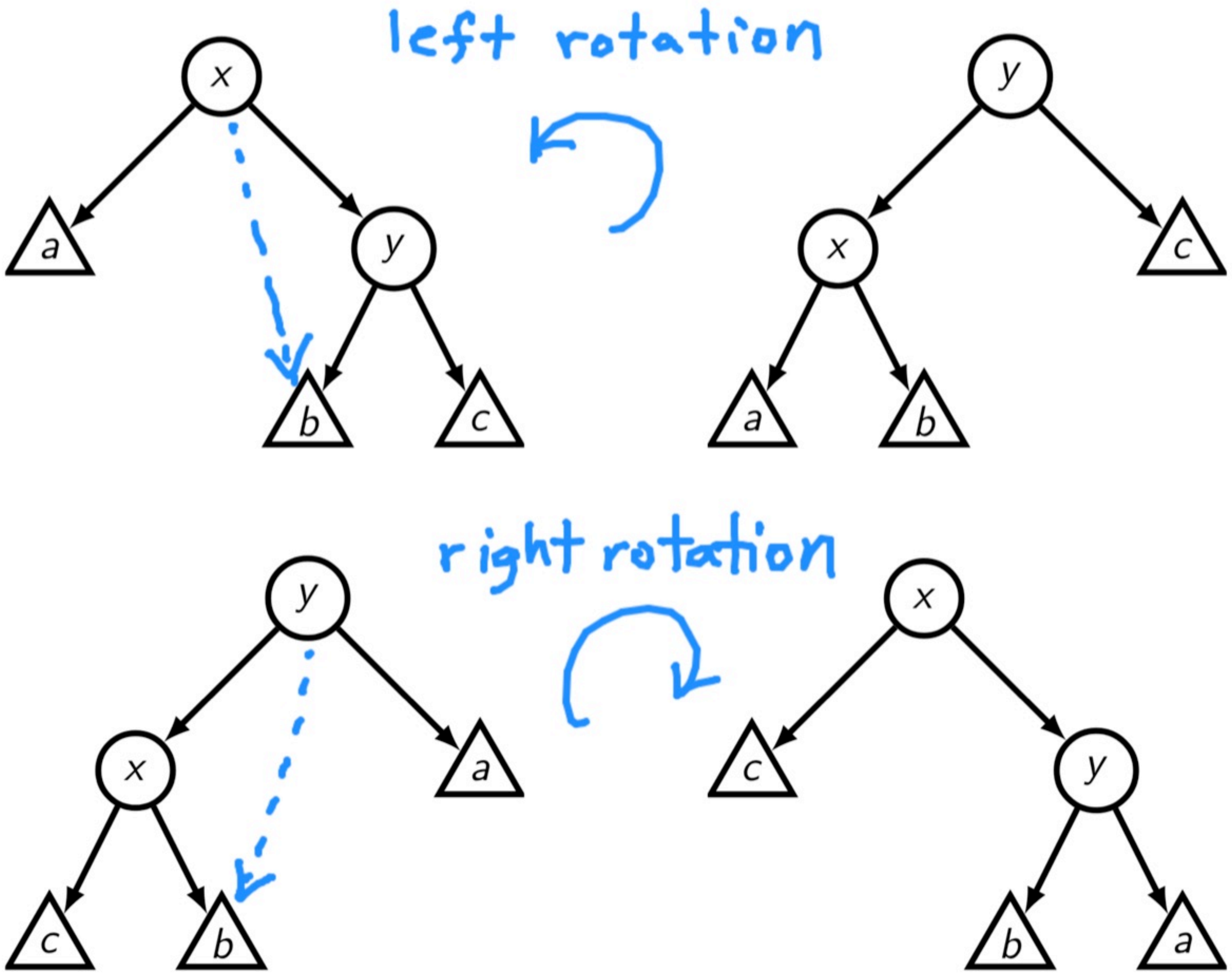
And then print the balance factor information to each node when printing out the tree.

```java
public int getBalanceFactor() {
  int hL = (left == null) ? -1 : left.height;
  int hR = (right == null) ? -1 : right.height;
  return hL - hR;
}

public String toString() {
  return "(" + key.toString() + ": " + getBalanceFactor() + ")";
}
```

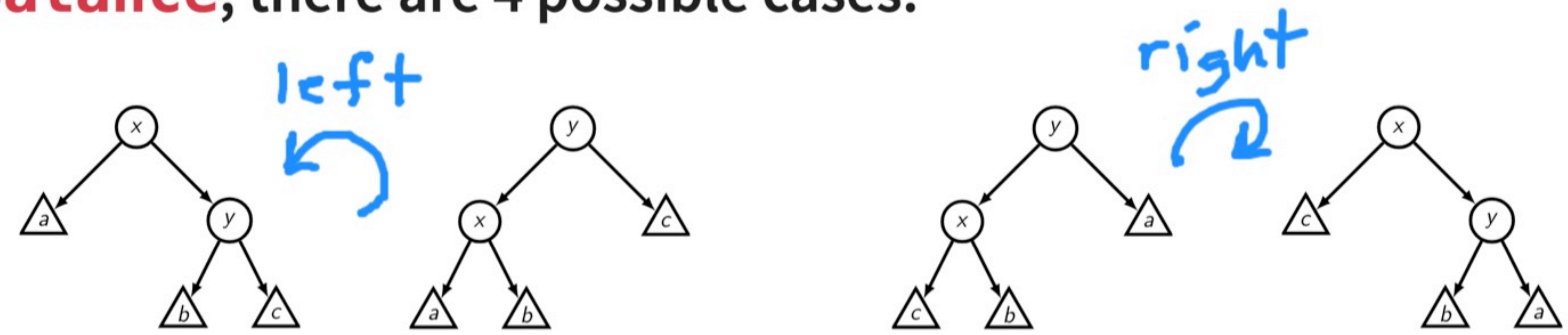


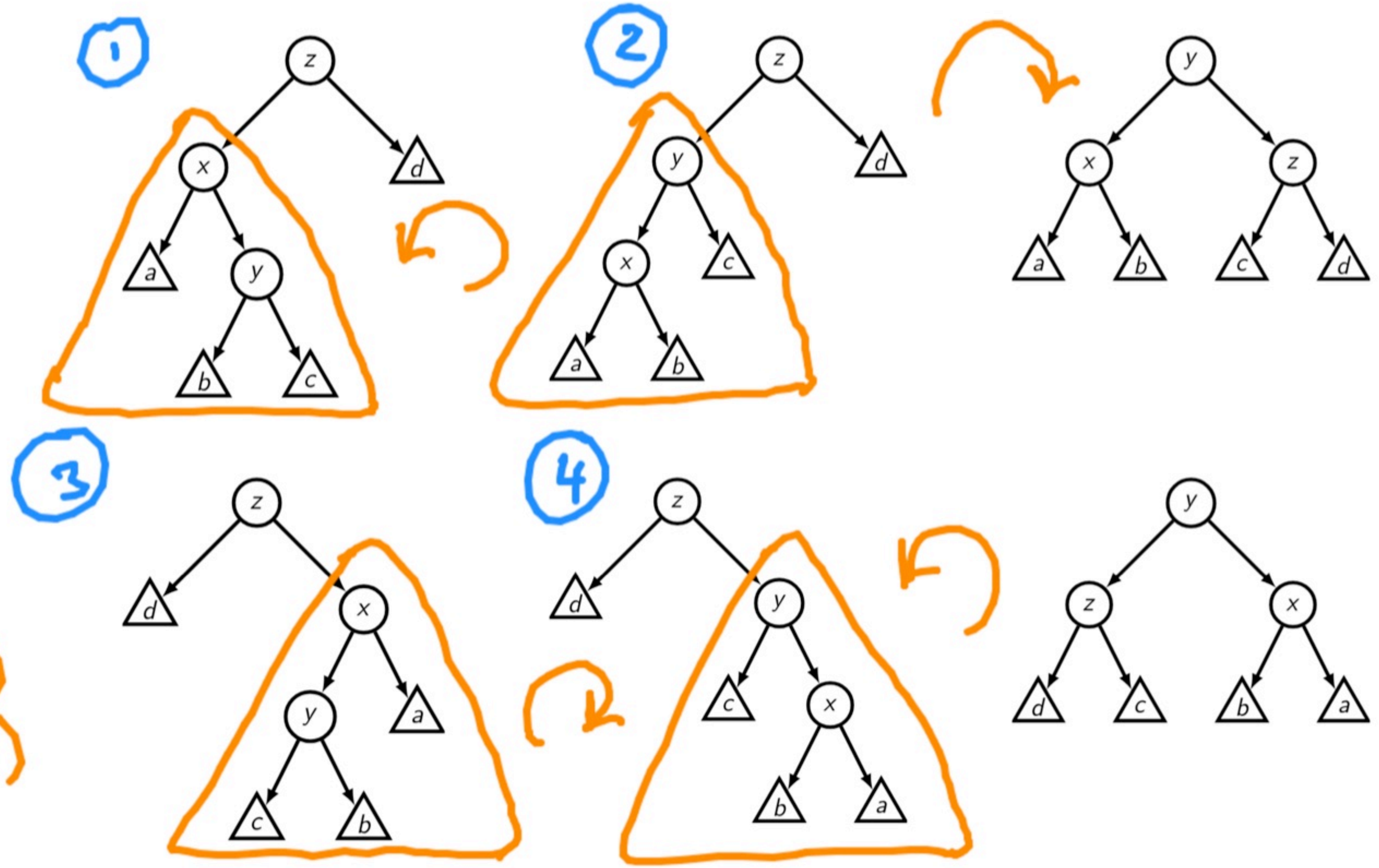
So what can we do with this balance factor? (which can be computed from the updated `height` during every `add` or `remove`).

ROTATIONS



To **balance**, there are 4 possible cases:



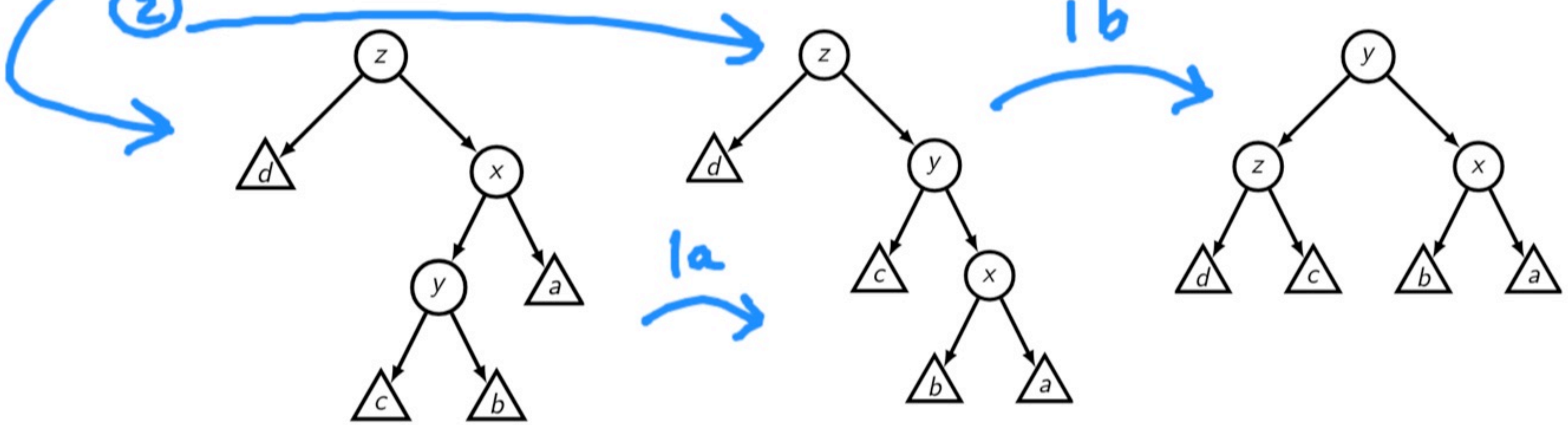


bf(z) > 1
bf(x) < 0
rotateL(x)
rotateR(z)

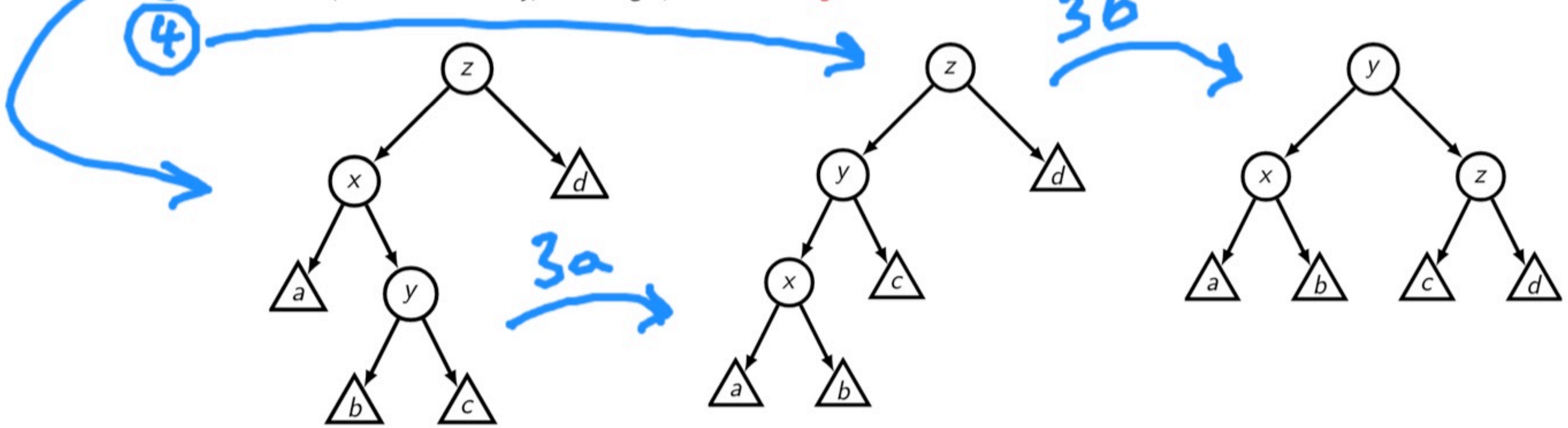
bf(z) < -1
bf(x) > 0
rotateR(x)
rotateL(z)

Maintaining an Adelson-Velsky-Landis (AVL) self-balancing binary search tree.

1. Perform `add`/`remove` as in usual BST operations.
2. Update node heights and calculate `node` balancing factor `bf(node)`.
3. `balance(node):`
   - `if bf(node) < -1:` (tree is right-heavy)
     - 1 `if bf(node.right) > 0` (right tree is left-heavy): **rotate right-left**, i.e. `rotateRight(node.right)` 1a then `rotateLeft(node)` 1b.
     - 2 `else` (right tree is right heavy): **rotate left**, i.e. `rotateLeft(node)`.

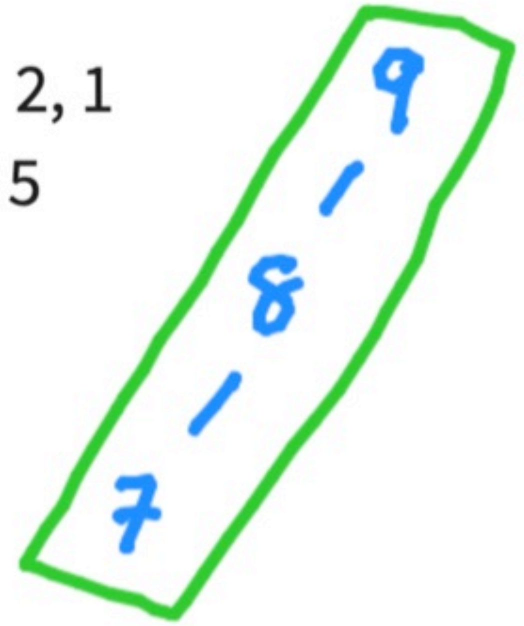


   - `if bf(node) > 1:` (tree is left-heavy)
     - 3 `if bf(node.left) < 0` (left tree is right-heavy): **rotate left-right**, i.e. `rotateLeft(node.left)` 3a then `rotateRight(node)` 3b.
     - 4 `else` (left tree is left-heavy): **rotate right**, i.e. `rotateRight(node)`.

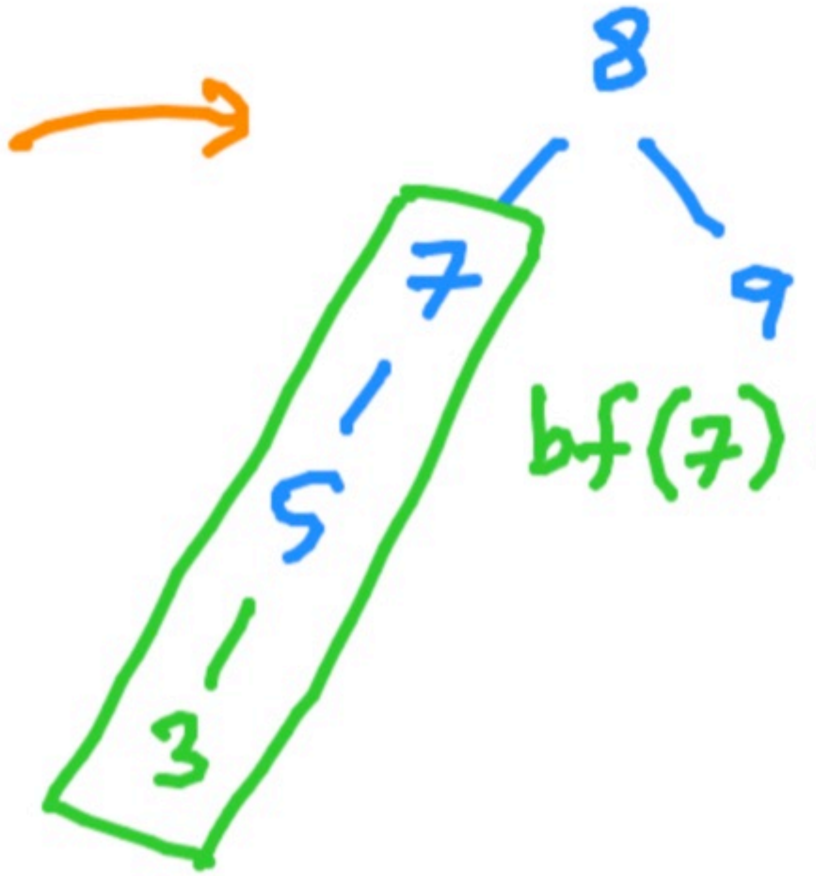


Practice: draw the AVL tree resulting from these operations.

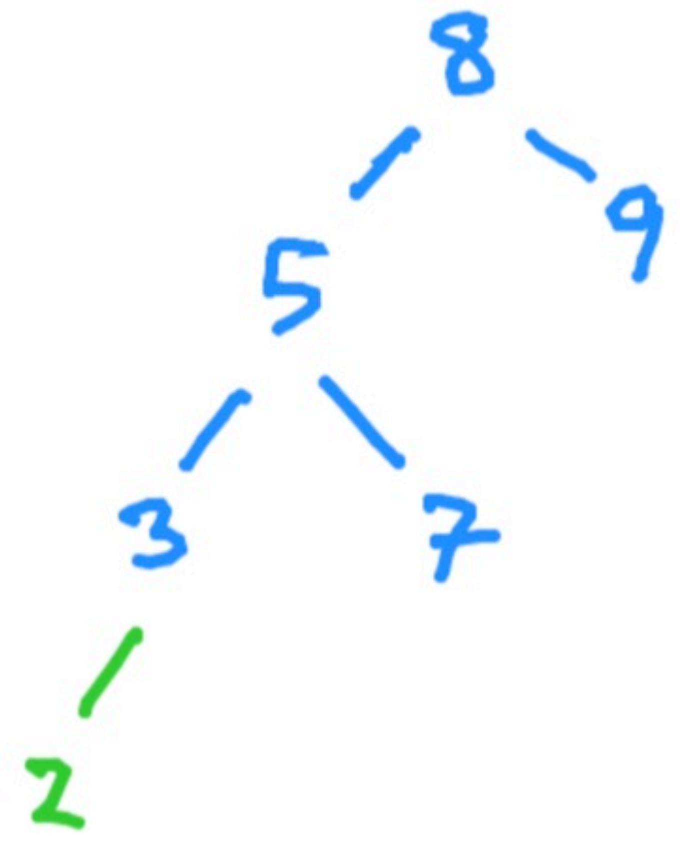
- add: 9, 8, 7, 5, 3, 2, 1
- then remove 7, 5



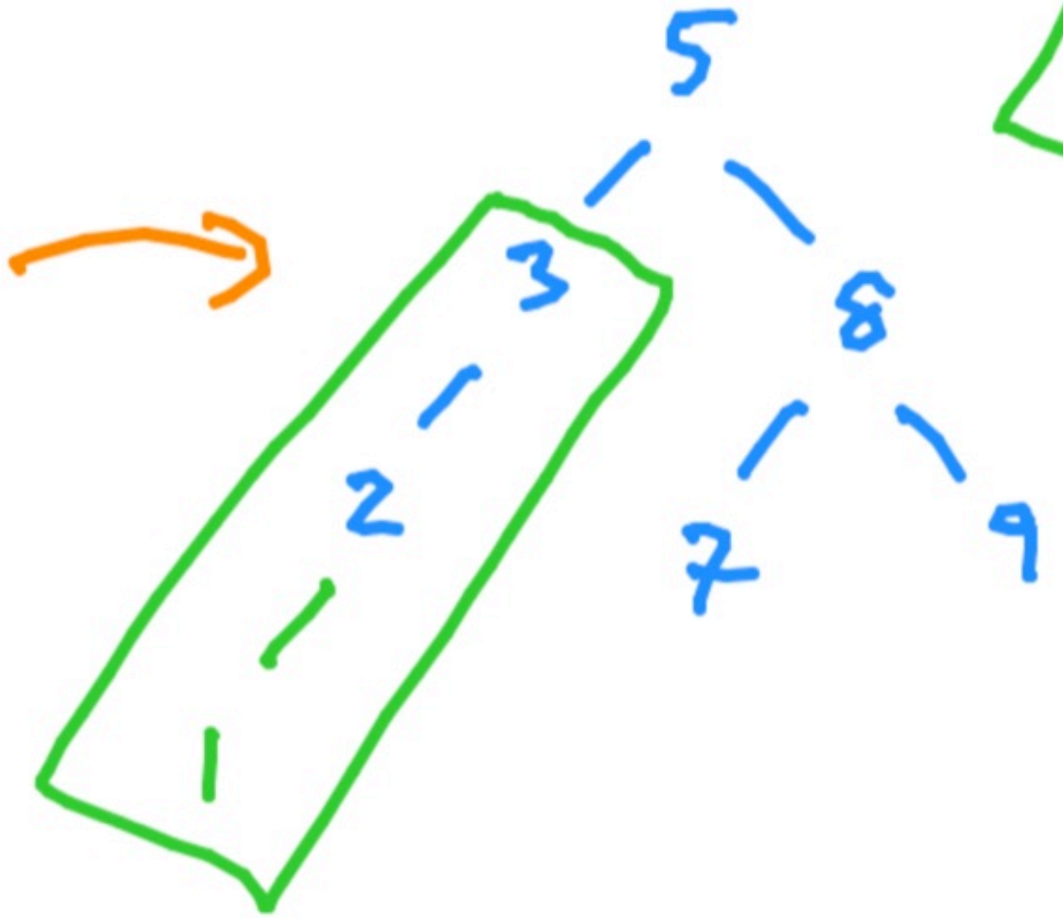
bf(9) = 1 − (−1)
= 2
bf(8) = 0 − (−1)
= 1

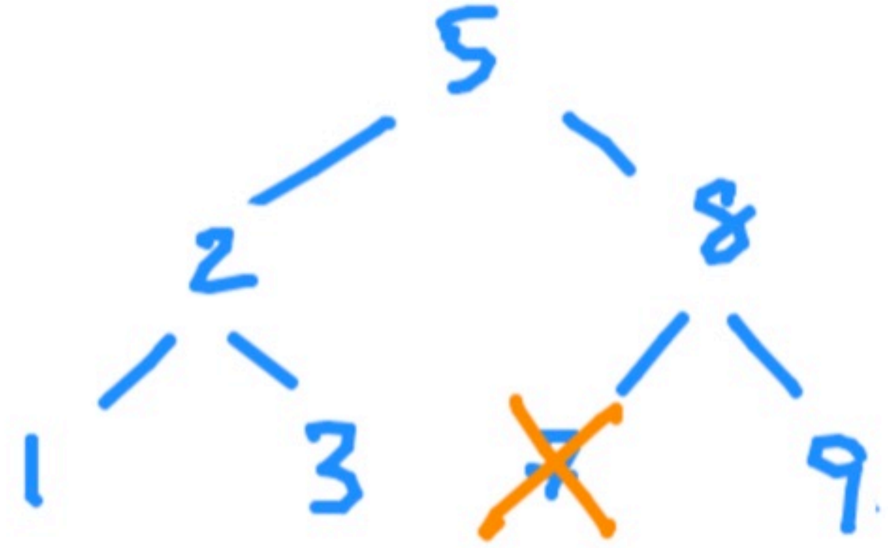


bf(7) = 1 − (−1)
= 2



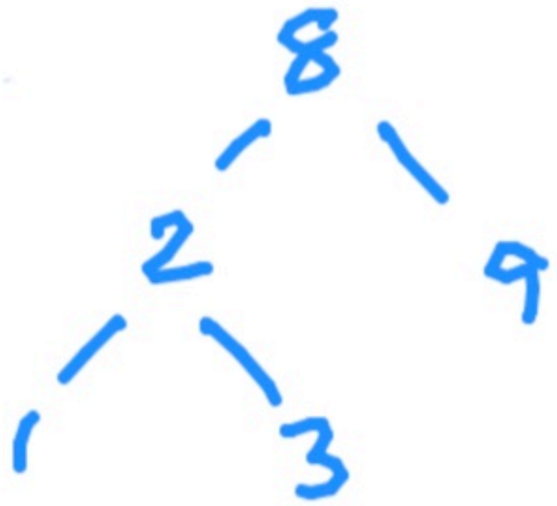
bf(8): 2 − 0
= 2
bf(5) = 1 − 0
= 1





remove(7)

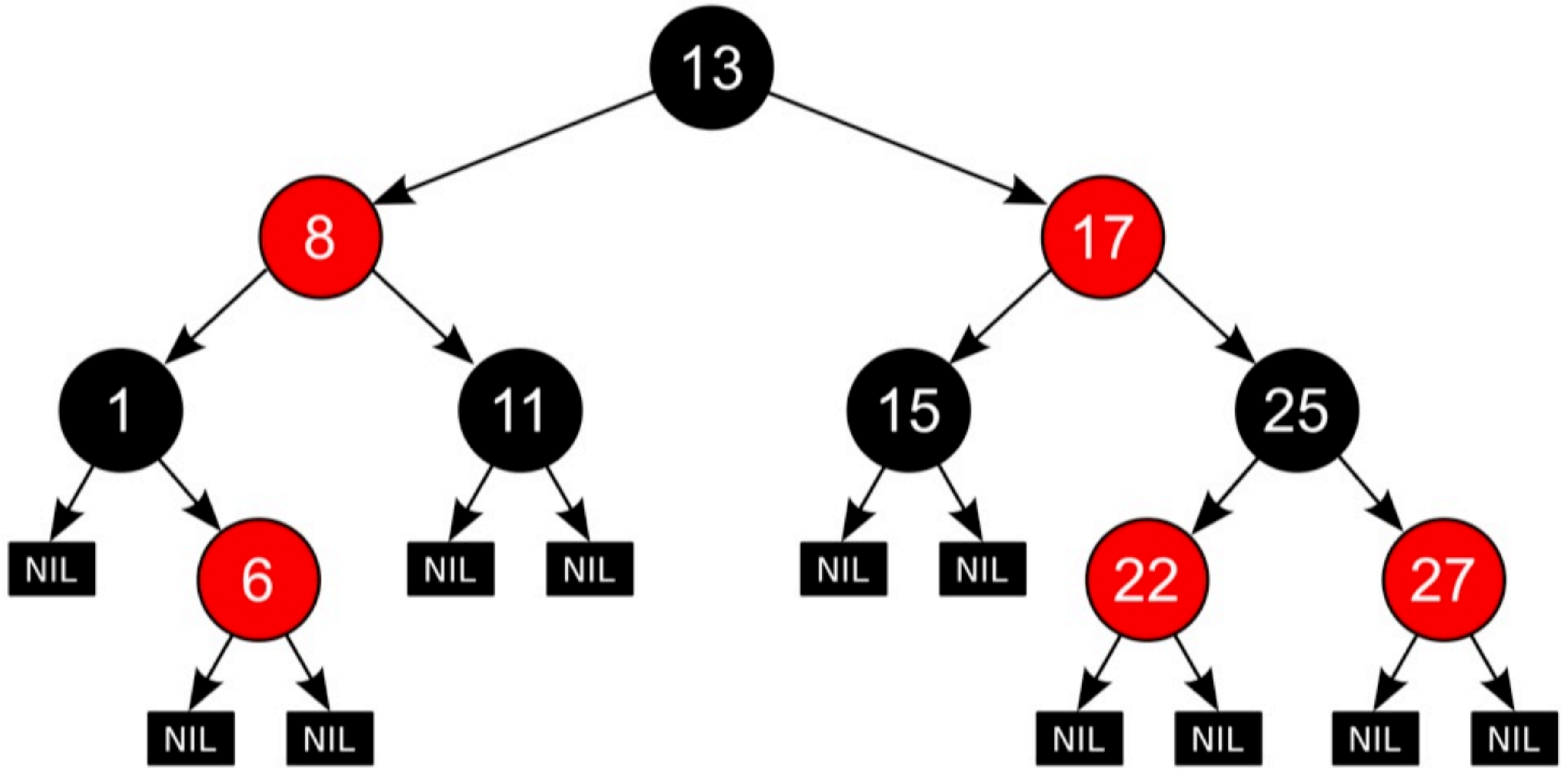
to remove(5)
minNode is 8:



Additional notes:

★ Registration, waivers!

- Homework 7 due on Thursday 4/17: implement a max-heap.
- There is also an optional Homework 7 challenge problem about Huffman compression.
- `TreeSet` and `TreeMap` use another type of balanced BST called a **Red-Black Tree**: extra bit (red or black) stored at every node, still uses rotations to balance (less than AVL to `add` but AVL tree will be more height-balanced so `contains` will be faster) .



source: Wikipedia (https://en.wikipedia.org/wiki/Red%E2%80%93black_tree)