
Middlebury

# CSCI 201: Data Structures

## Fall 2024

---

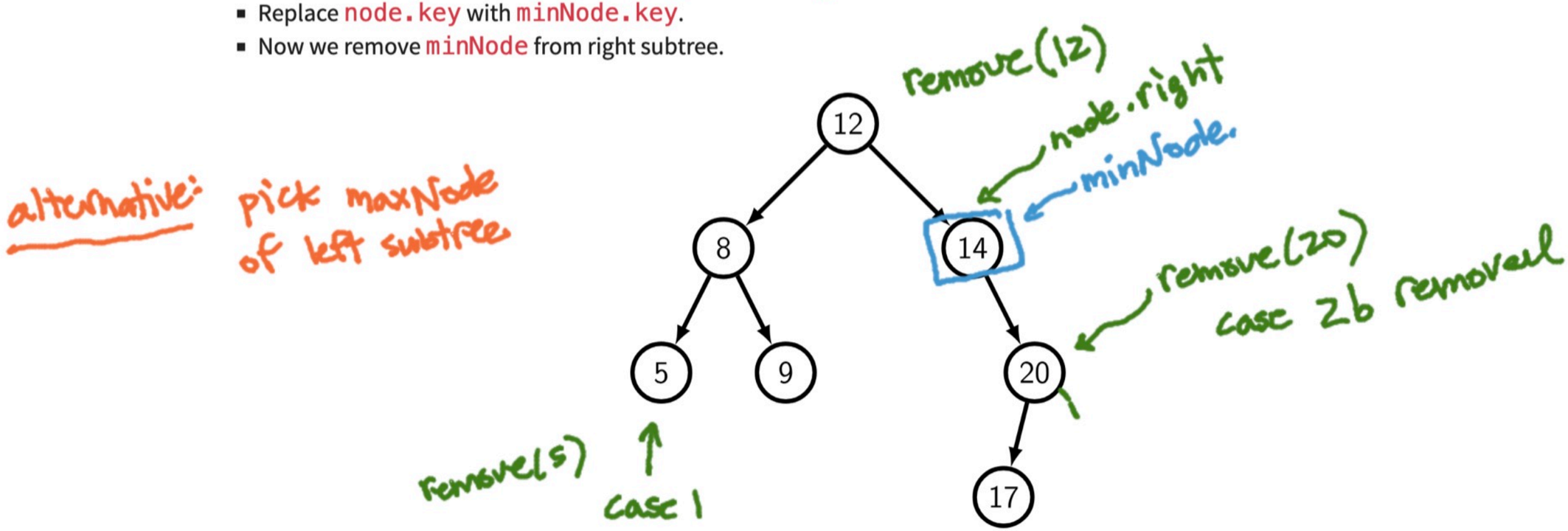
# Lecture 9R: Balanced Trees

# 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.

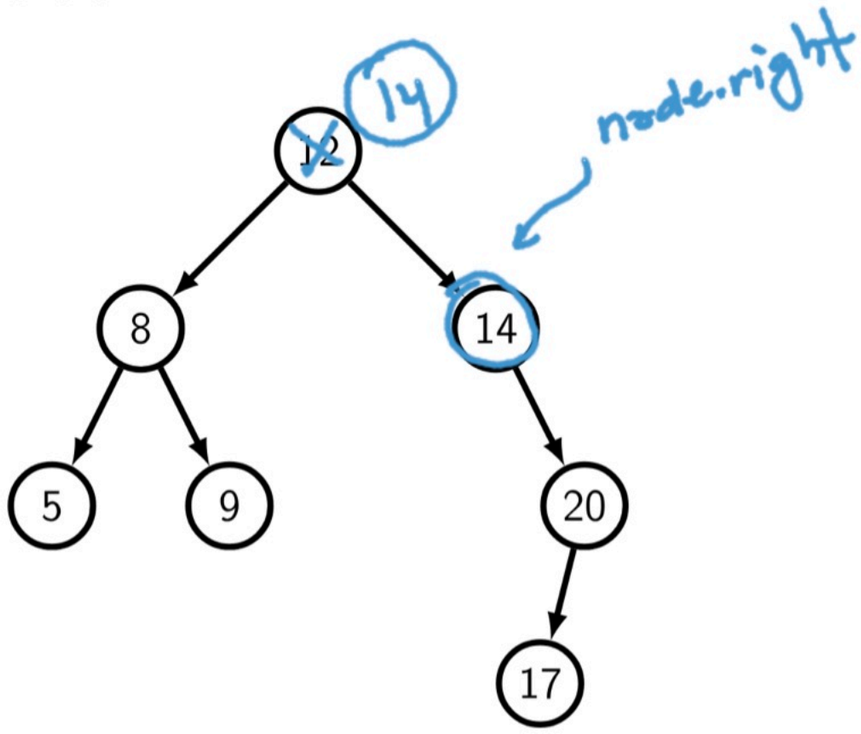
```java
public void remove(E key) {
  root = remove(root, key);
}

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;
}
```

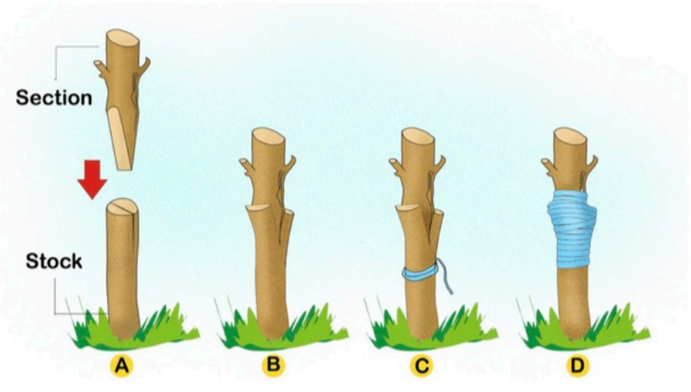
a.) find minNode → right subtree (method)

b.) set node.key = minNode.key

c.) call remove on right subtree

then save into node.right.



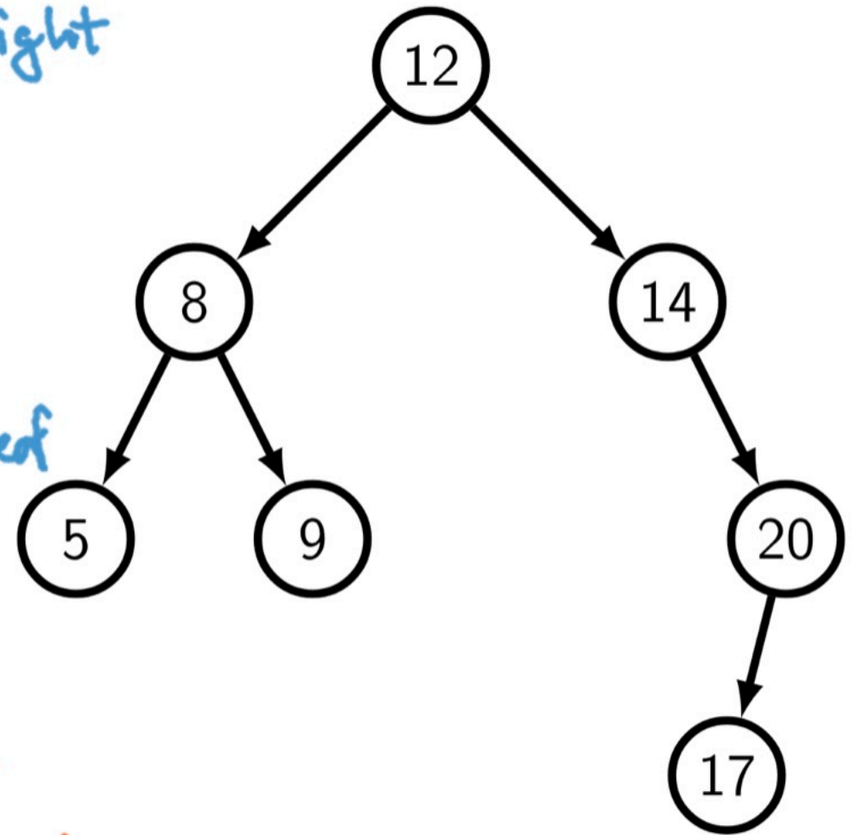



**Last time we talked about `contains`, `add`, `remove`.**
**What is the runtime complexity?**

contains: search left or right
O(height)

add: adds to leaf
O(height)

remove: can go down to leaf
O(height)




height can be n or log(n)
usually O(log n) if balanced

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

## 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);
  }
}
```

O(logn)

## 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. M2
- **Heap** (min or max): complete, every node value > child node values (for max-heap). M2
- **Binary search tree**: left subtree keys < right subtree keys of every node. M1, M2.

# 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);
```

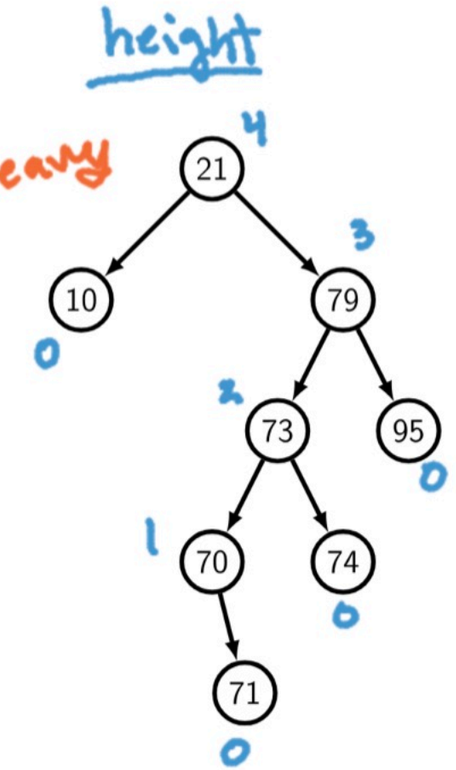

```
└─(0)
  └─(1)
    └─(2)
      └─(3)
        └─(4)
          └─(5)
            └─(6)
              └─(7)
                └─(8)
                  └─(9)
                    └─(10)
                      └─(11)
                        └─(12)
                          └─(13)
                          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

# 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

right-heavy

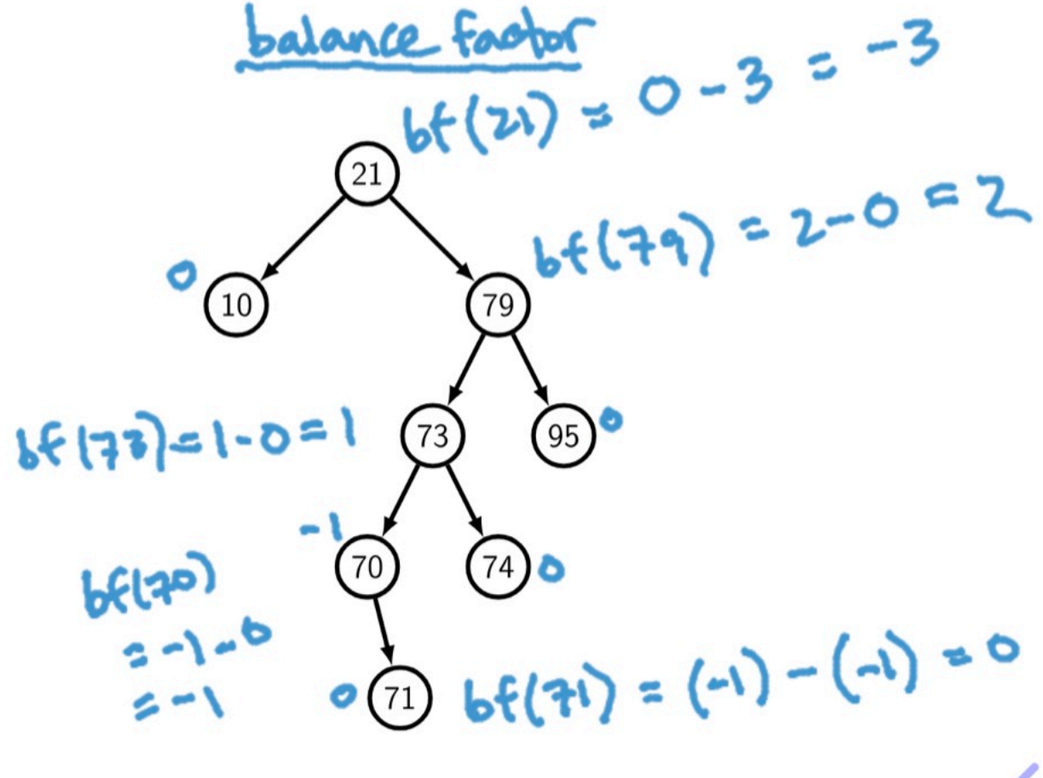



AVL trees:

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

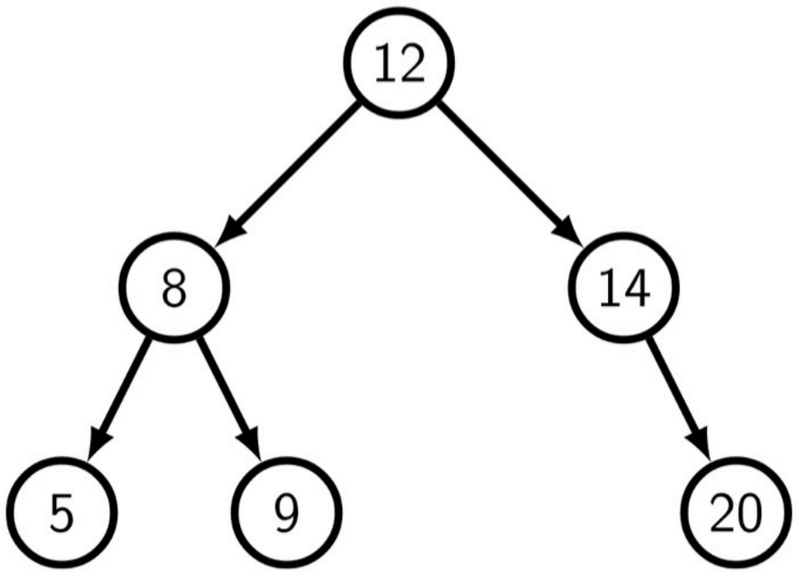
balance factor

bf(21) = 0 - 3 = -3



# 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;

  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);
  }
}
```

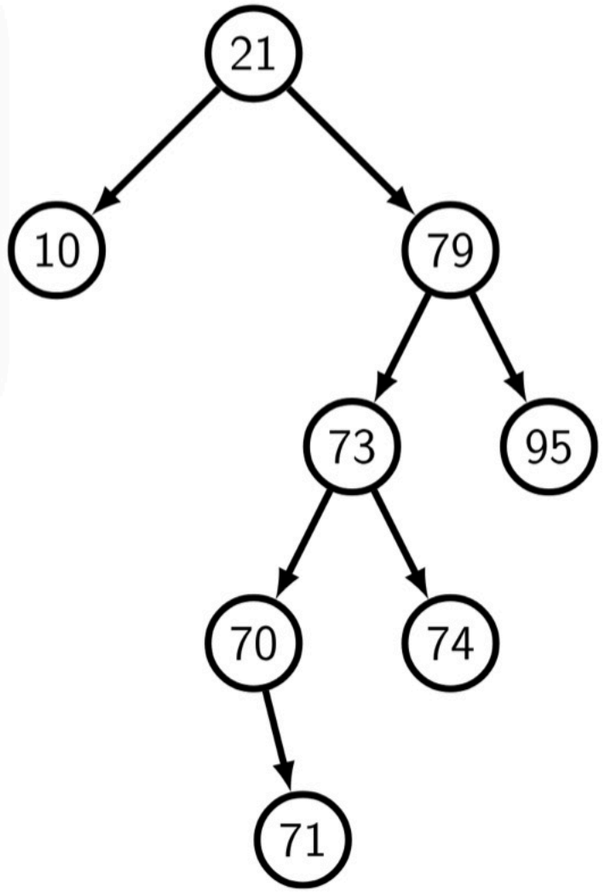
```java
public void calculateHeight() {
  int hL = (left == null) ? -1 : left.height;
  int hR = (right == null) ? -1 : right.height;
  height = 1 + Math.max(hL, hR);
}
```

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

bf = height of left − height of right.

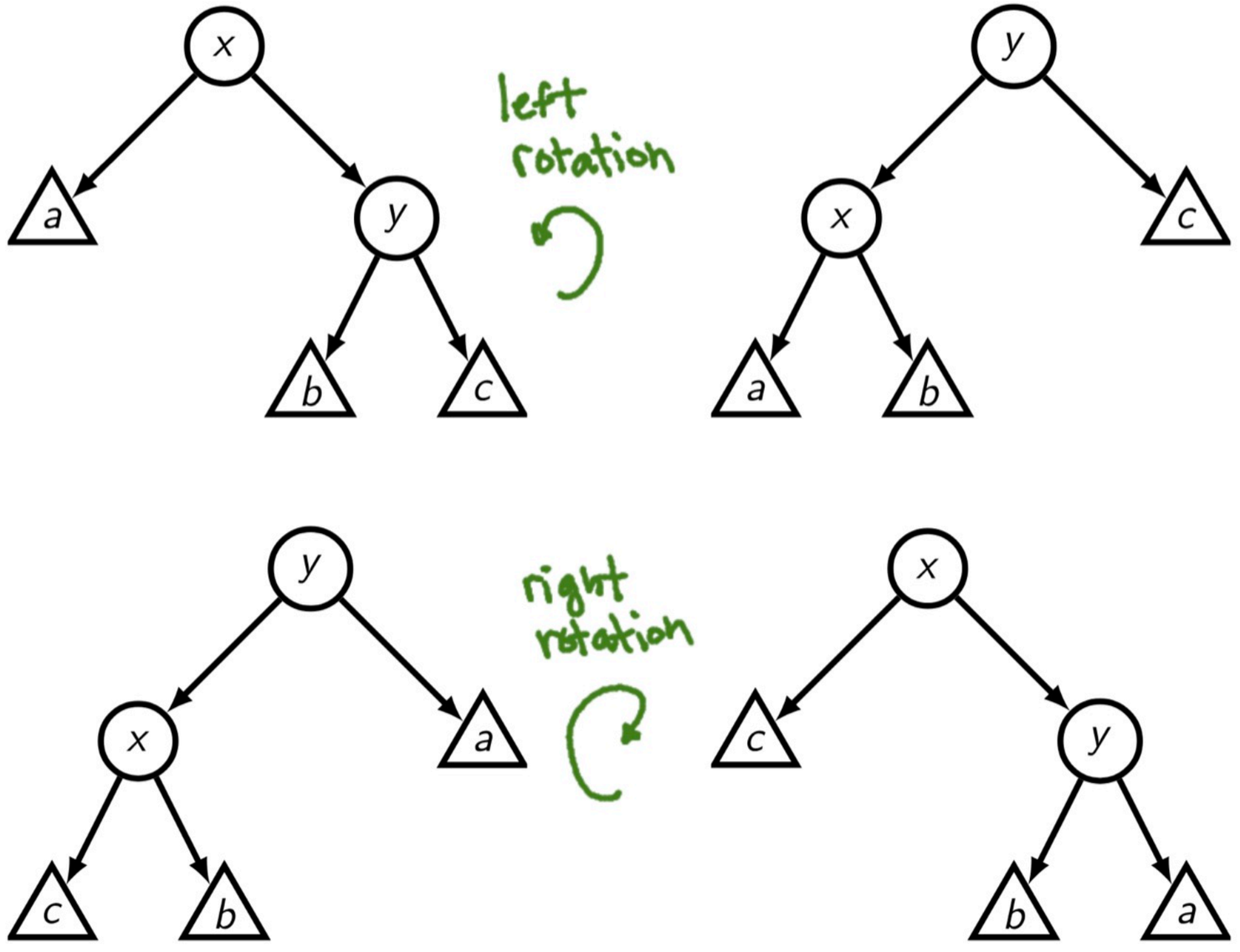
And then print the balance factor information to each node when printing out the tree.

```
1 public int getBalanceFactor() {
2   int hL = (left == null) ? -1 : left.height;
3   int hR = (right == null) ? -1 : right.height;
4   return hL - hR;
5 }
6
7 public String toString() {
8   return "(" + key.toString() + ": " + getBalanceFactor() + ")";
9 }
```



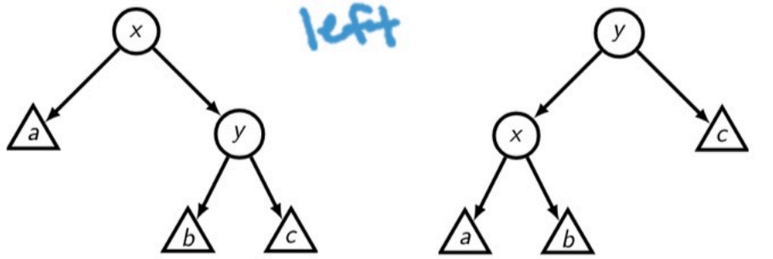
So what can we do with this balance factor? (which can be computed from the updated `height` during every `add` or `remove`).

## ROTATIONS



## To `balance`, there are 4 possible cases:

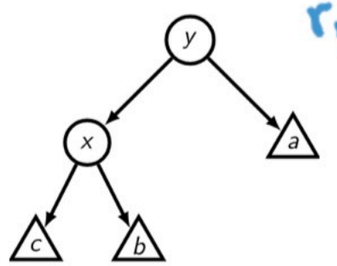
left

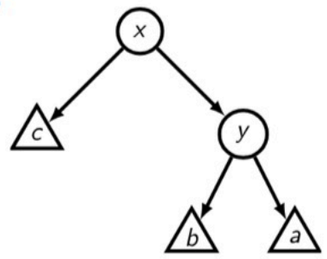






right



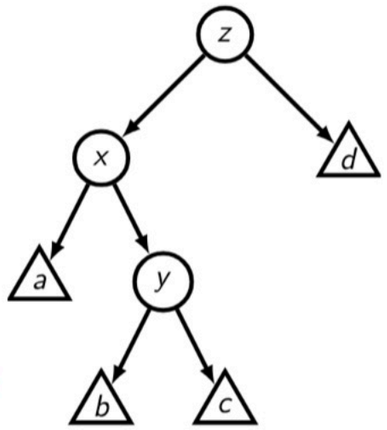

bf(z) > 1

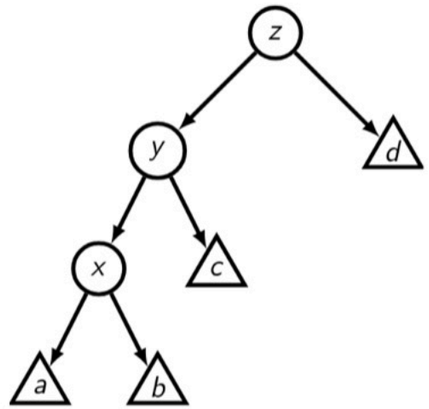
bf(x) < 0

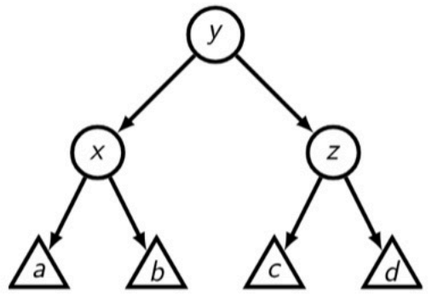
↳ z.left.

rotateLeft(x)

then rotateRight(z)



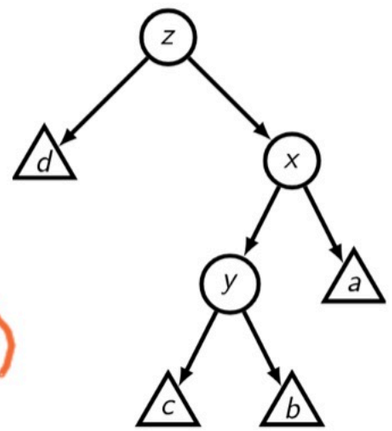

bf(z) < -1

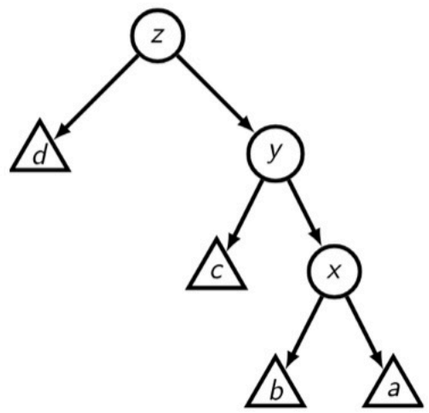
bf(x) > 0

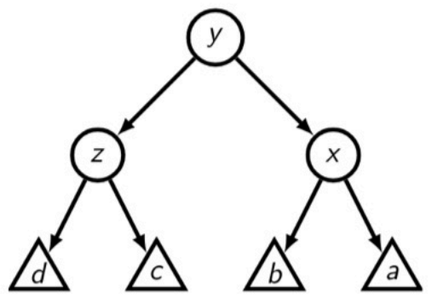
↳ z.right

rotateRight(x)

then rotateLeft(z)

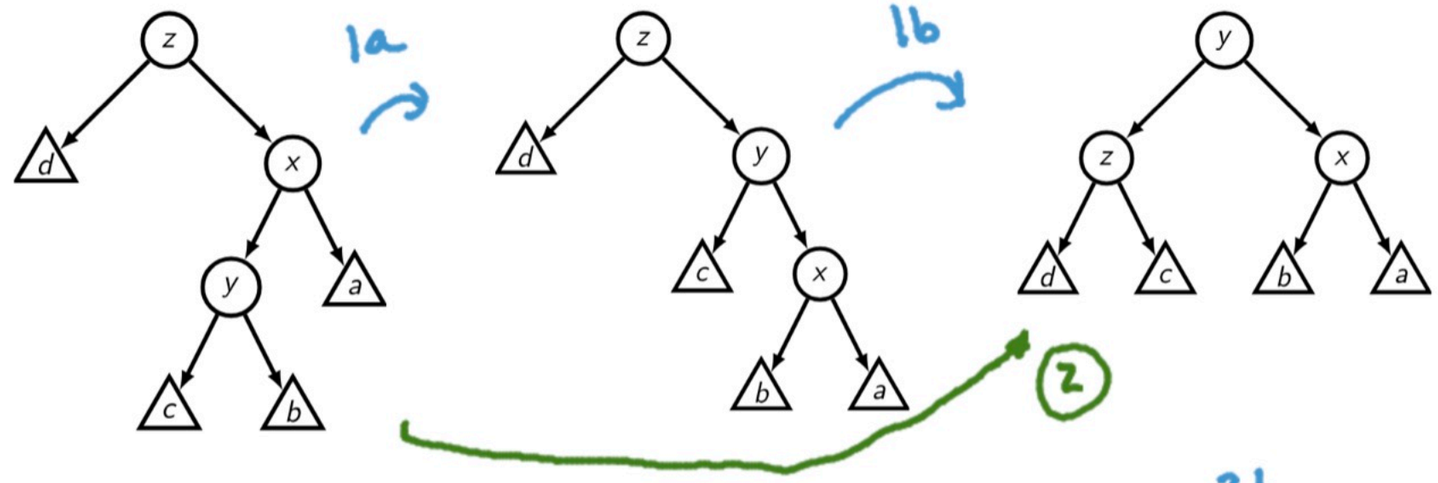


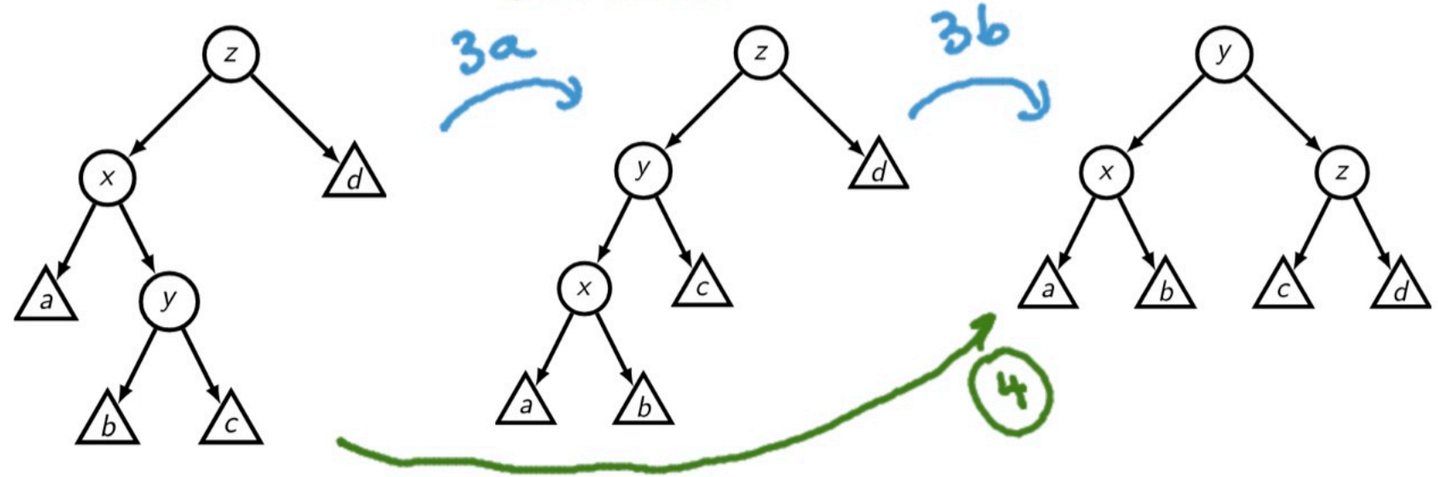



# Maintaining an Adelson-Velsky-Landis (AVL) self-balancing binary search tree.

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

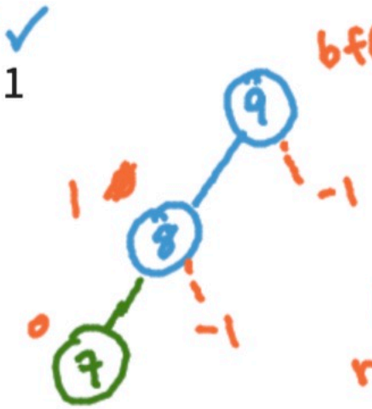



   - `if bf(node) > 1:` (tree is left-heavy)
     - `if bf(node.left) < 0` (left tree is right-heavy): **rotate left-right**, i.e. `rotateLeft(node.left)` then `rotateRight(node)`.
     - `else` (left tree is left-heavy): **rotate right**, i.e. `rotateRight(node)`.



# Practice: draw the AVL tree resulting from these operations.

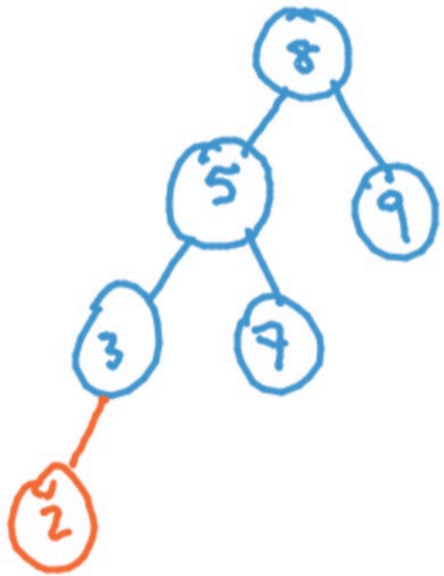
- add: 9, 8, 7, 5, 3, 2, 1
- then remove 7, 5




bf(9) = 0 − (−1) = +1

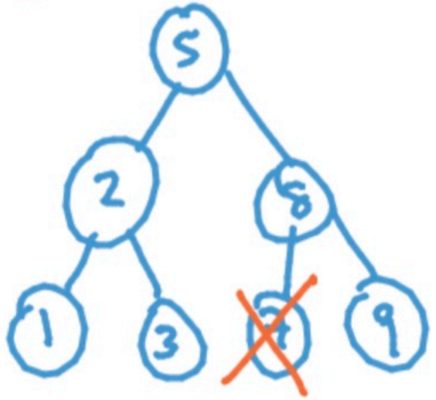
now
bf(9) = 1 − (−1) = 2
bf(8) = 0 − (−1) = 1
rotate right


bf(7) = $h_5 - h_{null}$
= 1 − (−1)
= 2
rotate right




bf(8): $h_5 - h_9$
= 2 − 0 = 2
bf(5): $h_3 - h_7$
= 1 − 0 = 1
rotate right

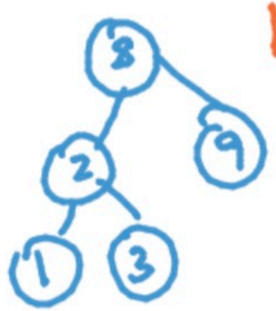

rotate right




remove(7)
bf(5) = $h_2 - h_8$
= 1 − 1 = 0

remove(5):

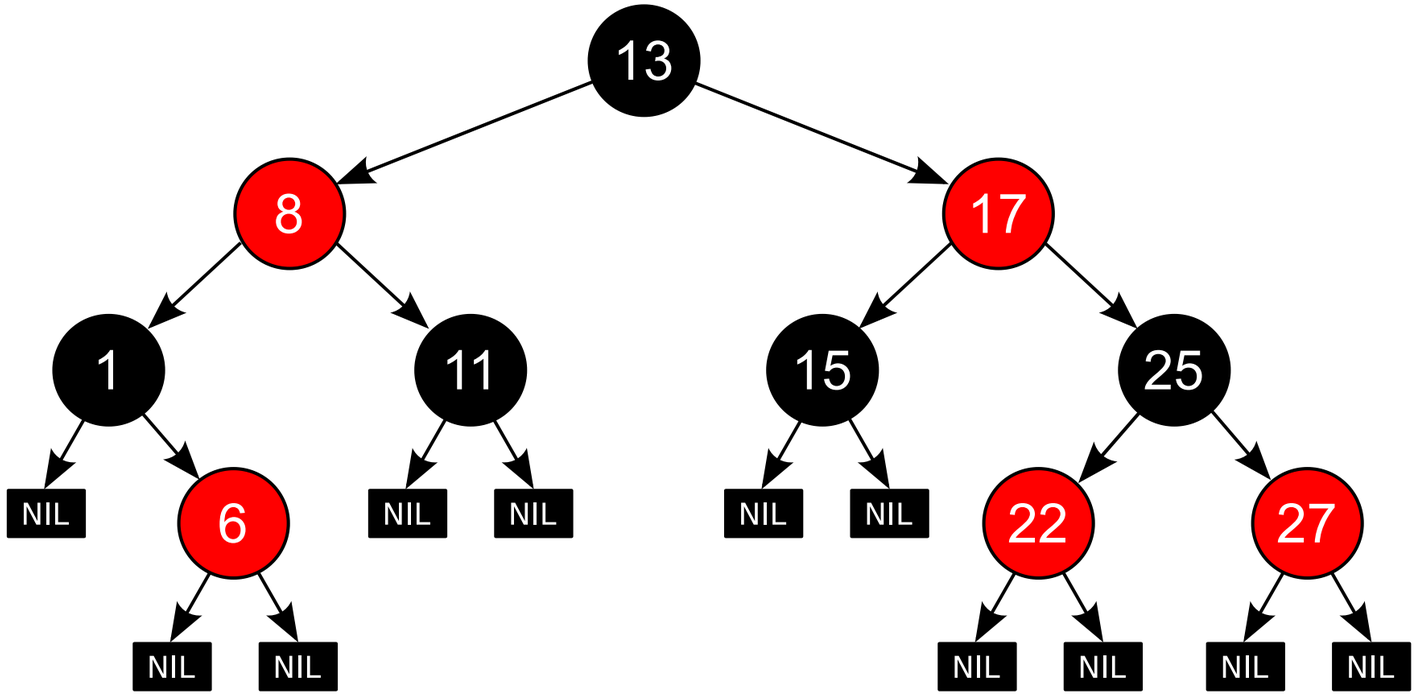
case 3
min node of R subtree = 8




bf(8) = $h_2 - h_9$
= 1 − 0 = 1

# Additional notes:

- **Homework 7** due on Friday 11/8: 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) .
- Submit ET 9R by the end of today.




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