Middlebury

CSCI 201: Data Structures

Fall 2025

Lecture 9R: Balanced Trees

Goals for today:

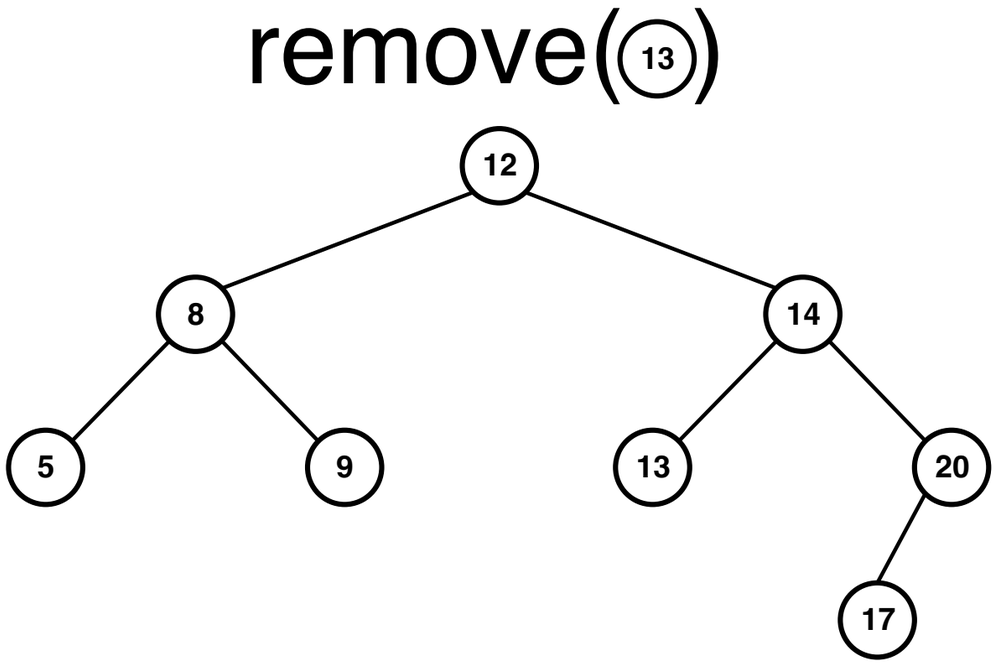
- Discuss how to `remove` from a BST, and how to implement a `remove` method.
- Analyze the complexity of `contains`, `add`, `remove` in a BST.
- **Motivation for balancing:** what happens if we insert keys in a BST in a certain way?
- Calculate and save the height of a node when it's added.
- Calculate 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.

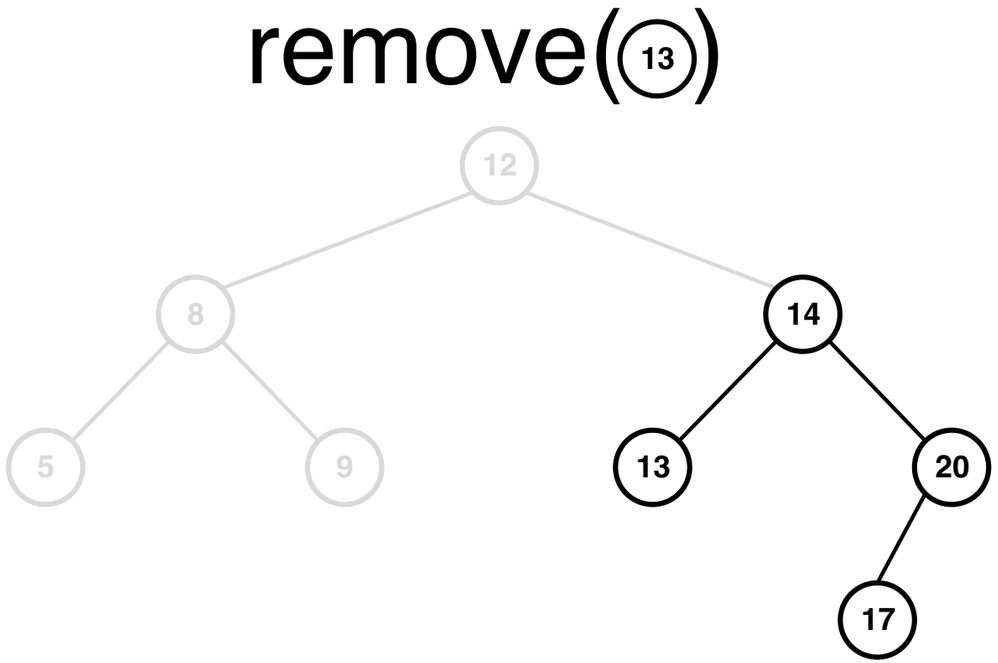
Removing an item/`key` from a BST:

- 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.



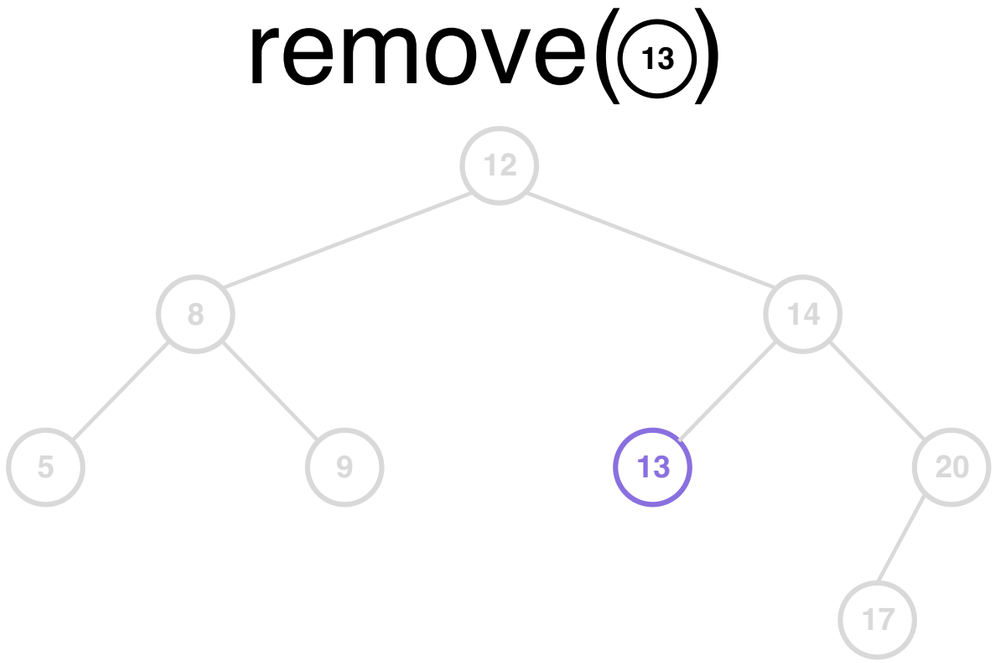
Removing an item/`key` from a BST:

- 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.



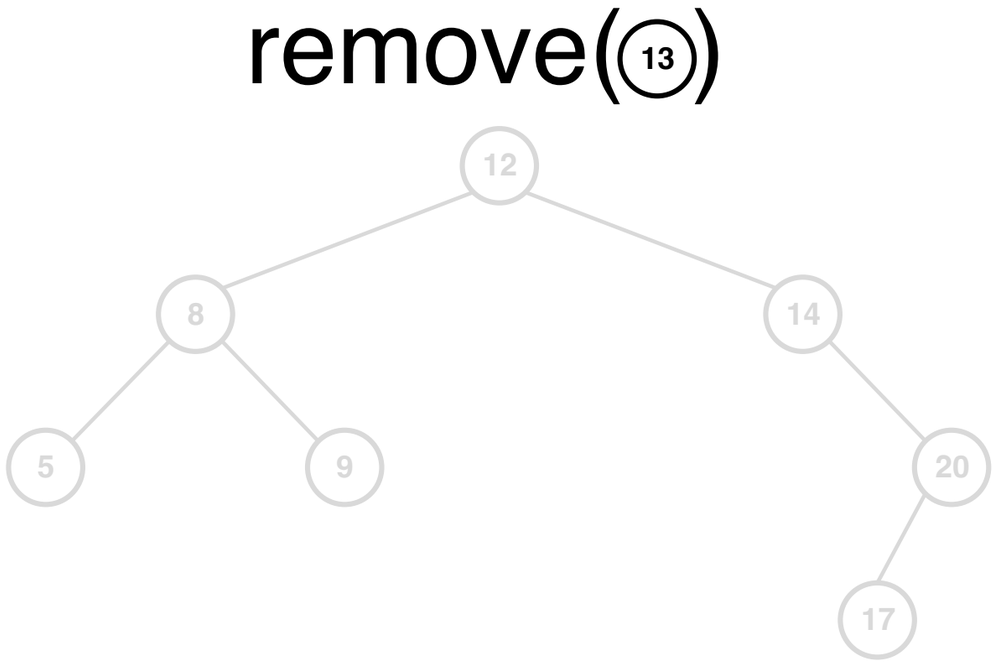
Removing an item/`key` from a BST:

- 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.



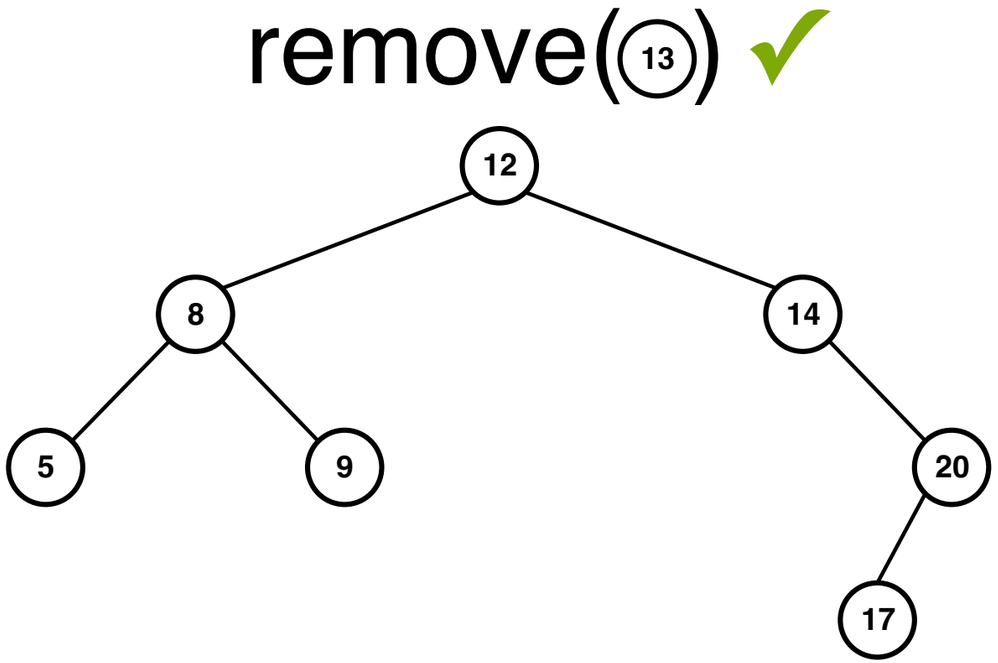
Removing an item/`key` from a BST:

- 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.



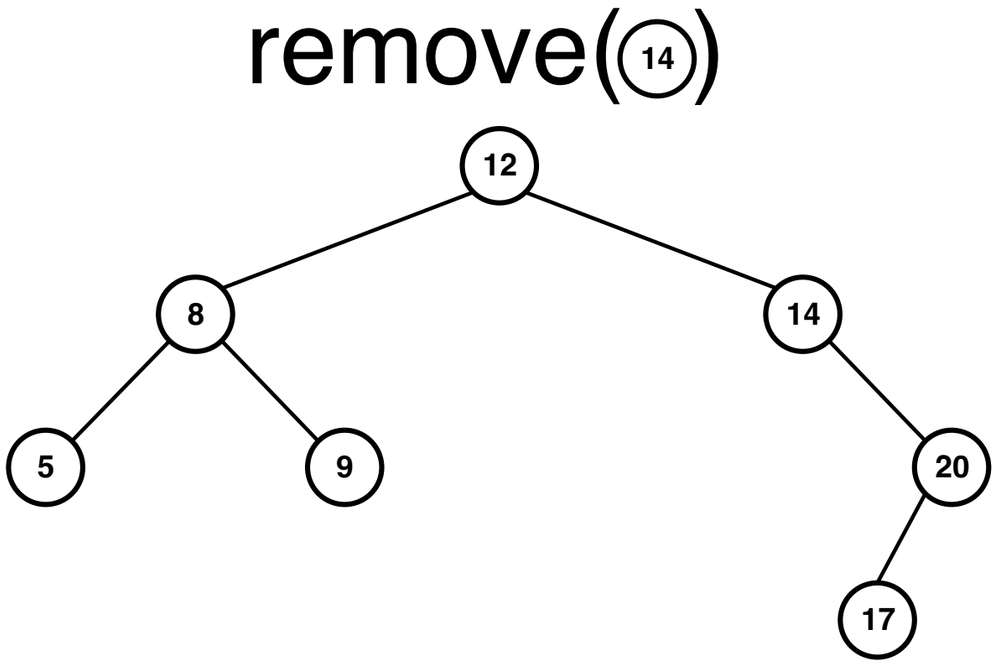
Removing an item/`key` from a BST:

- 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.



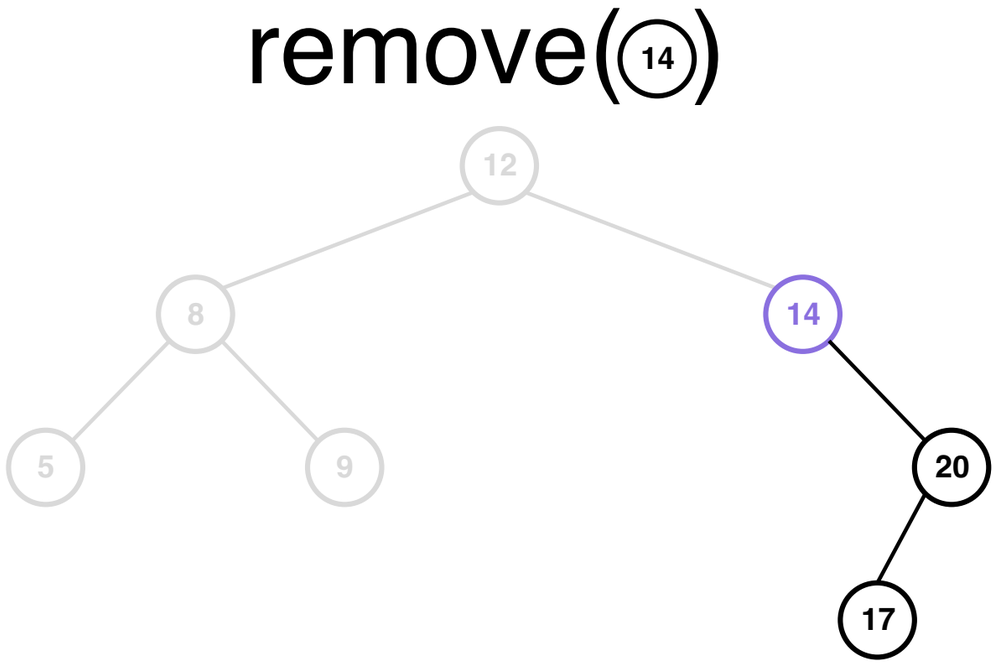
Removing an item/`key` from a BST:

- 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.



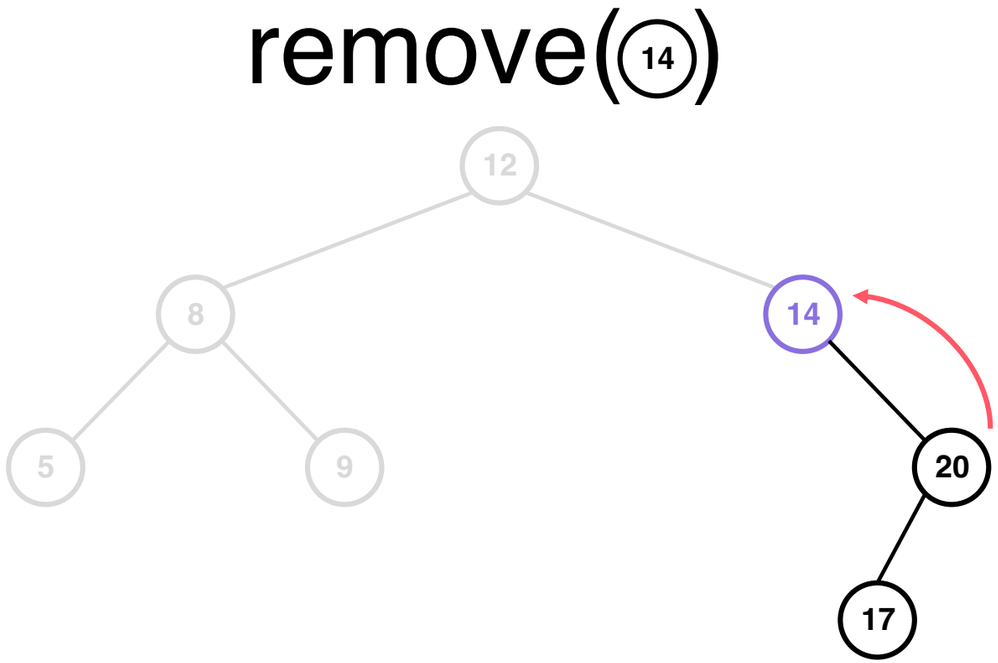
Removing an item/`key` from a BST:

- 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.



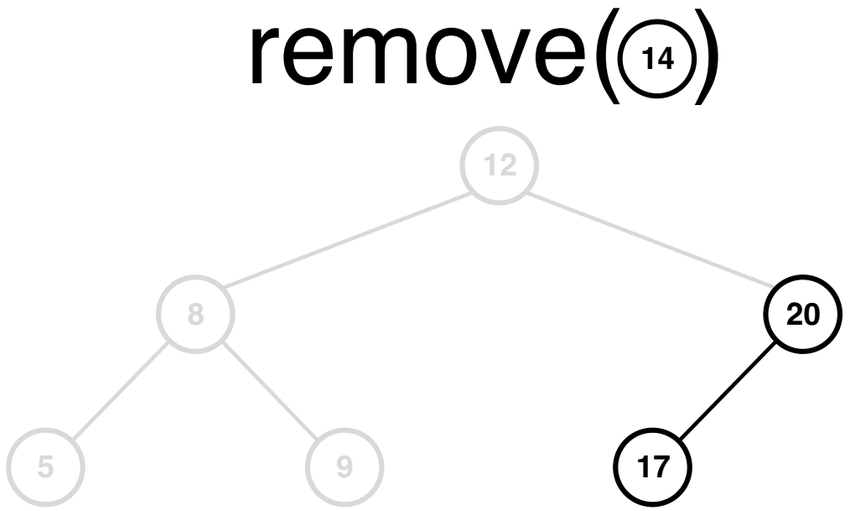
Removing an item/`key` from a BST:

- 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.



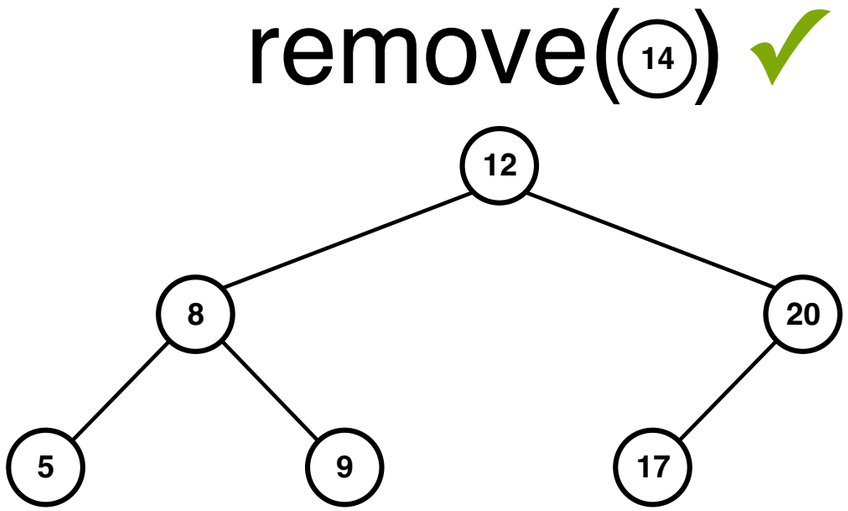
Removing an item/`key` from a BST:

- 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.



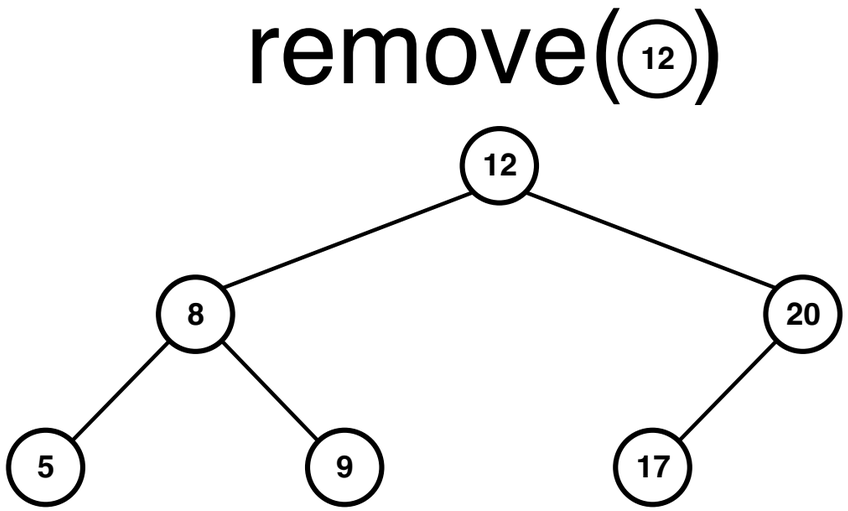
Removing an item/`key` from a BST:

- 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.



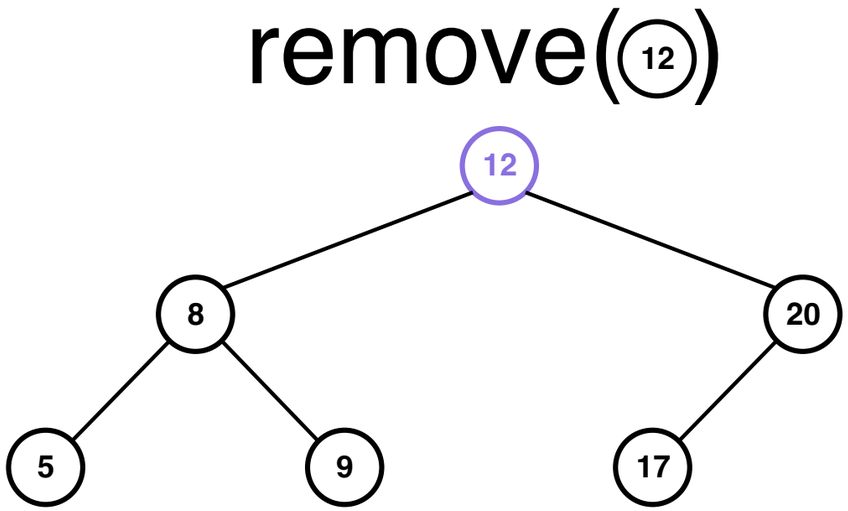
Removing an item/`key` from a BST:

- 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.



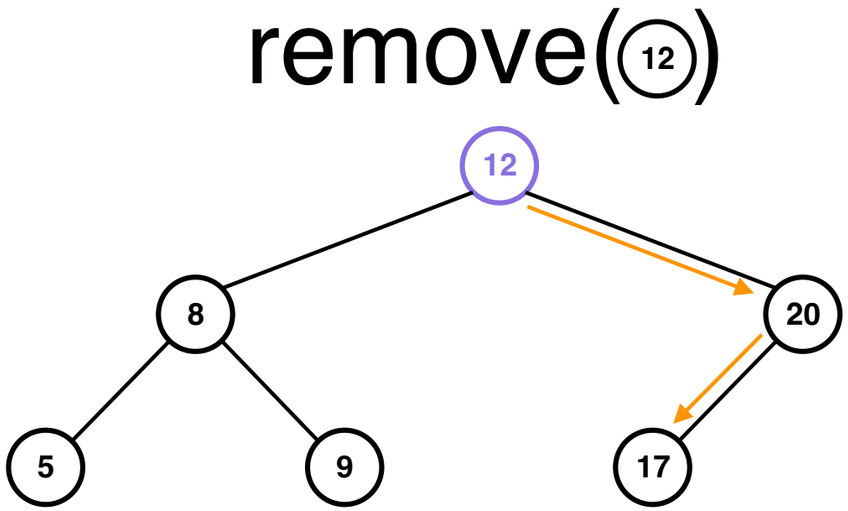
Removing an item/`key` from a BST:

- 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.



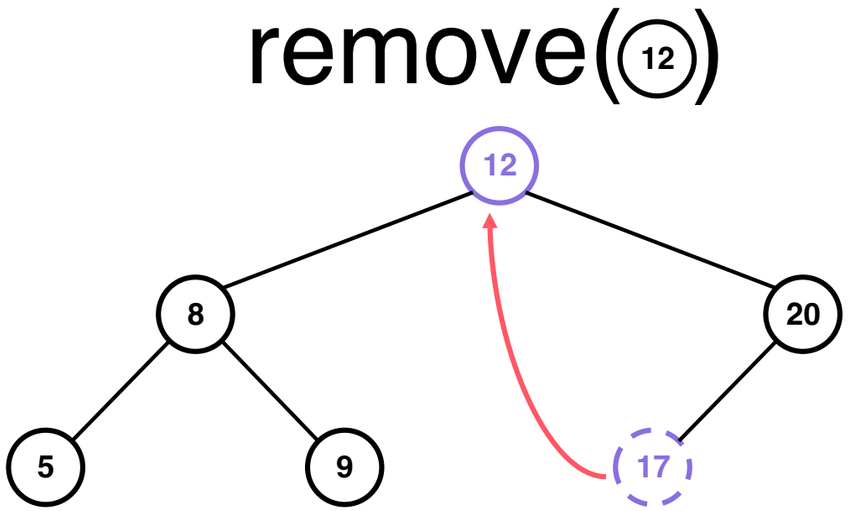
Removing an item/`key` from a BST:

- 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.



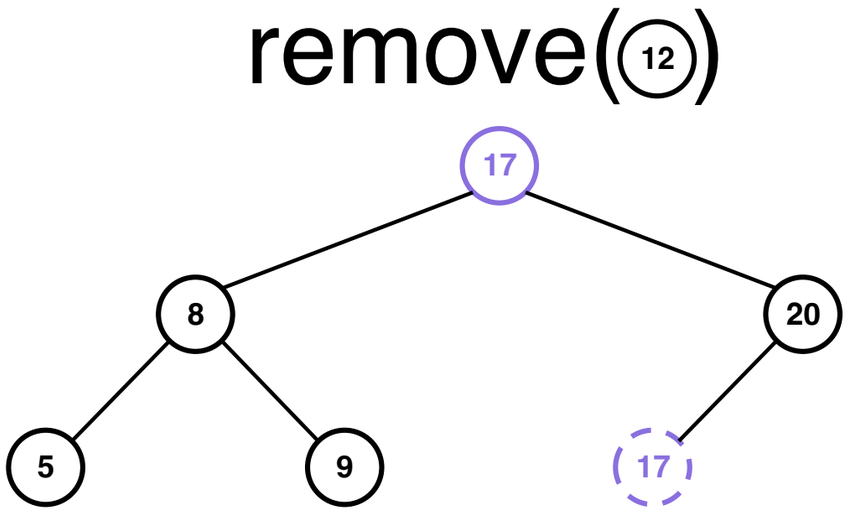
Removing an item/`key` from a BST:

- 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.



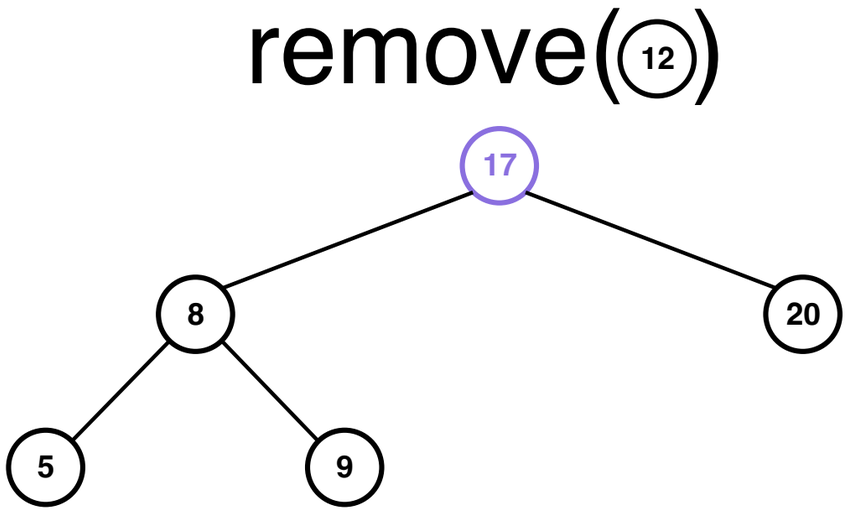
Removing an item/`key` from a BST:

- 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.



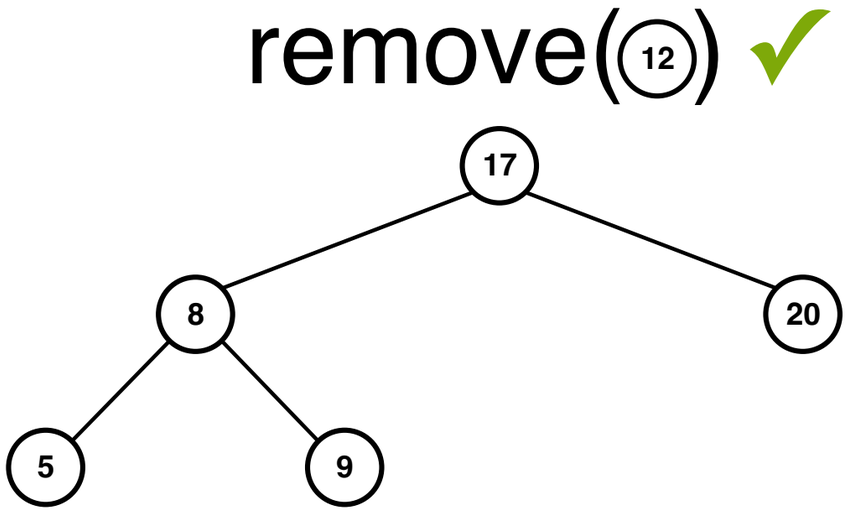
Removing an item/`key` from a BST:

- 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.



Removing an item/`key` from a BST:

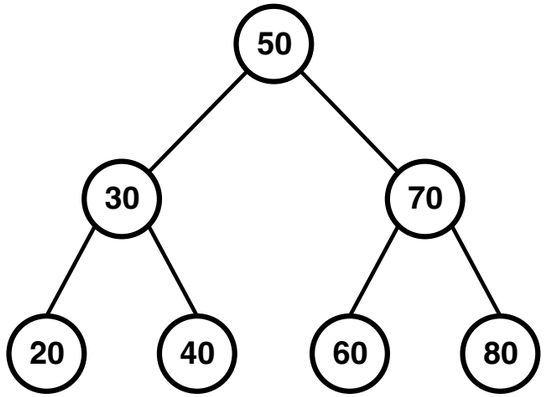
- Start at `node = root`.
- If `key < node.key`, need to remove node in left subtree (`node.left`).
- If `key > node.key`, need to remove node in right subtree (`node.right`).
- If `key == node.key`, three cases to consider:
    1. Is this a leaf? Delete node.
    2. (a) If `node.left == null`, "graft" `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.



Exercise: draw the tree after the following calls to `add` and `remove`.

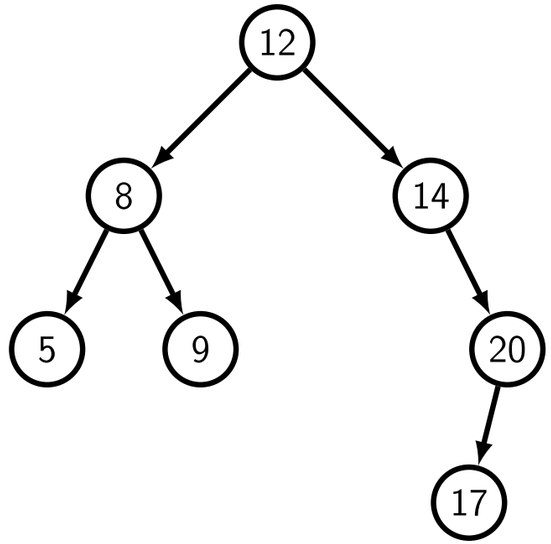
Work on the whiteboards in groups.

- `add`: 55, 65, 58, 75, 90
- then `remove`: 70, 60, 50



Implementing the `remove` method.

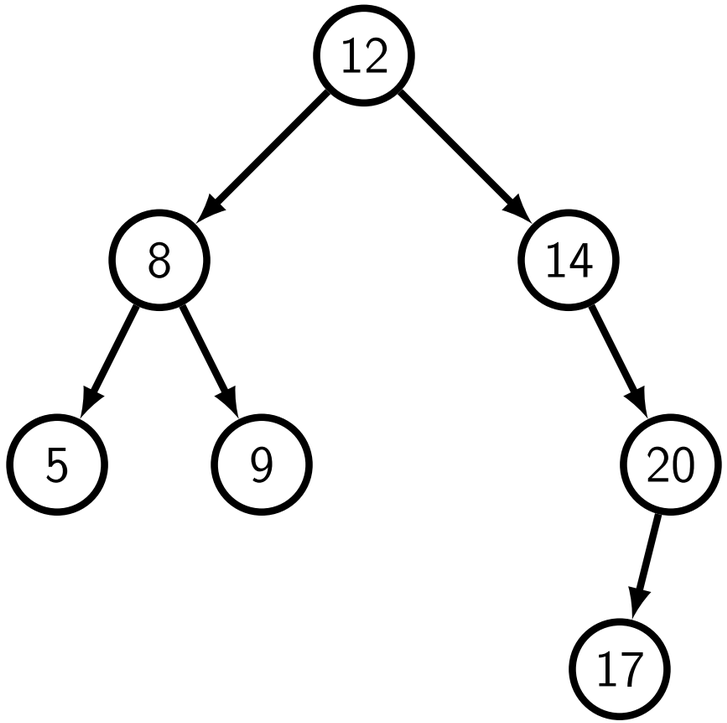
```java
public void remove(E key) {
  root = remove(root, key);
}

private Node<E> remove(Node<E> 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;
}
```



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.

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

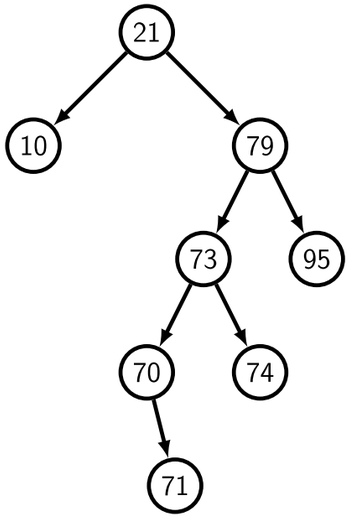


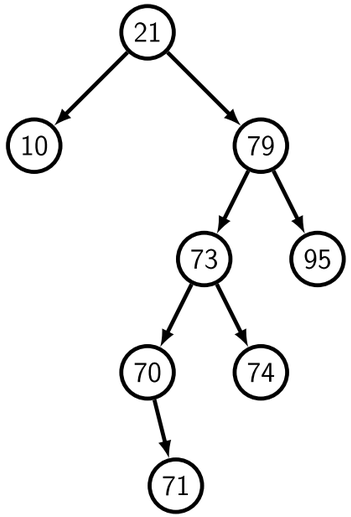
Motivation for balancing: what happens if we do this?

```
1 int n = 20;
2 for (int i = 0; i < n; i++) {
3   tree.add(i);
4 }
5 System.out.println(tree);
```

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

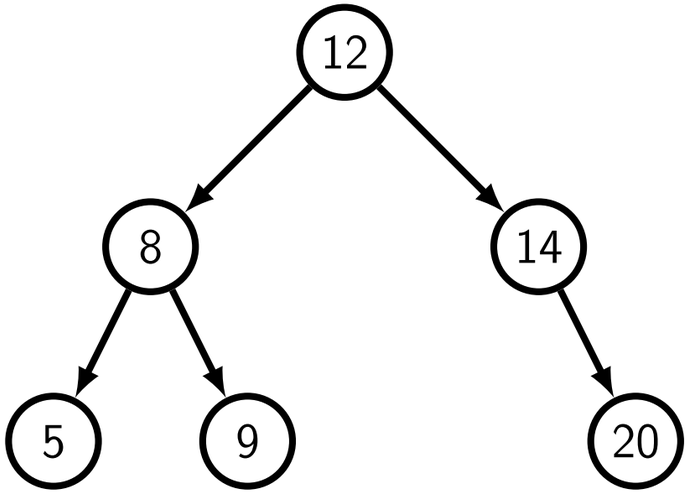
- Recall the **height** of a node (and tree): length of *longest* path from node to a leaf.
- **Balanced** means the height of the left and right subtrees differ by at most one.
- **Balance factor** (bf) of a node: `height(node.left) - height(node.right)`.
- height of a `null` node is `-1`.
- `bf < 0`: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





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
private class Node<E> {
  private E key;
  private Node<E> left;
  private Node<E> right;
  private int height;

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

  public void calculateHeight() {
    // TODO
  }
}

private Node<E> add(Node<E> 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<E> {
  private E key;
  private Node<E> left;
  private Node<E> right;
  private int height;

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

  public void calculateHeight() {
    // TODO
  }
}

private Node<E> add(Node<E> 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() {

  // attempt 1

}
```

```java
public void calculateHeight() {

  // attempt 2

}
```

```java
public void calculateHeight() {

  // attempt 3

}
```

Updating the height as we recurse back up the tree.

```java
class Node<E> {
  private E key;
  private Node<E> left;
  private Node<E> right;
  private int height;

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

  public void calculateHeight() {
    // TODO
  }
}

private Node<E> add(Node<E> 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() {

  // attempt 2

}
```

```java
public void calculateHeight() {

  // attempt 3

}
```

Updating the height as we recurse back up the tree.

```java
class Node<E> {
  private E key;
  private Node<E> left;
  private Node<E> right;
  private int height;

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

  public void calculateHeight() {
    // TODO
  }
}

private Node<E> add(Node<E> 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() {

  // attempt 3

}
```

Updating the height as we recurse back up the tree.

```java
class Node<E> {
  private E key;
  private Node<E> left;
  private Node<E> right;
  private int height;

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

  public void calculateHeight() {
    // TODO
  }
}

private Node<E> add(Node<E> 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;
}
```

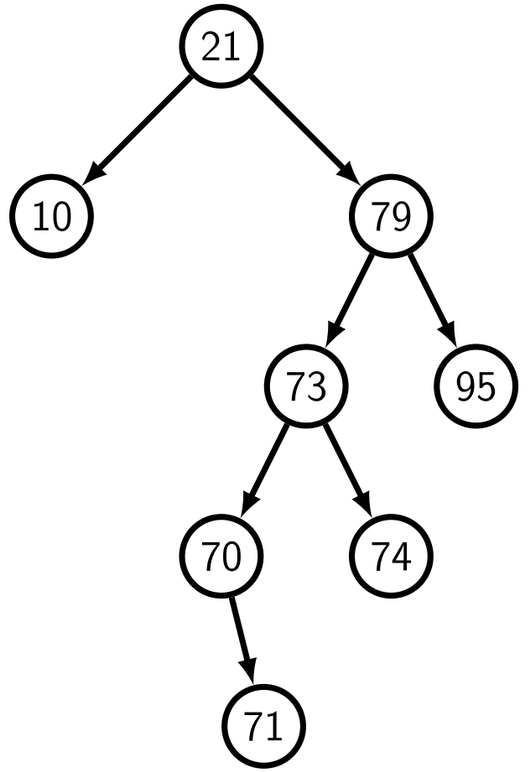
```java
public void calculateHeight() {
  if (left == null && right == null) {
    height = 0;
  } else {
    int hL = (left == null) ? 0 : node.left.calculateHeight();
    int hR = (right == null) ? 0 : node.right.calculateHeight();
    height = 1 + Math.max(hL, hR);
  }
}
```

```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.

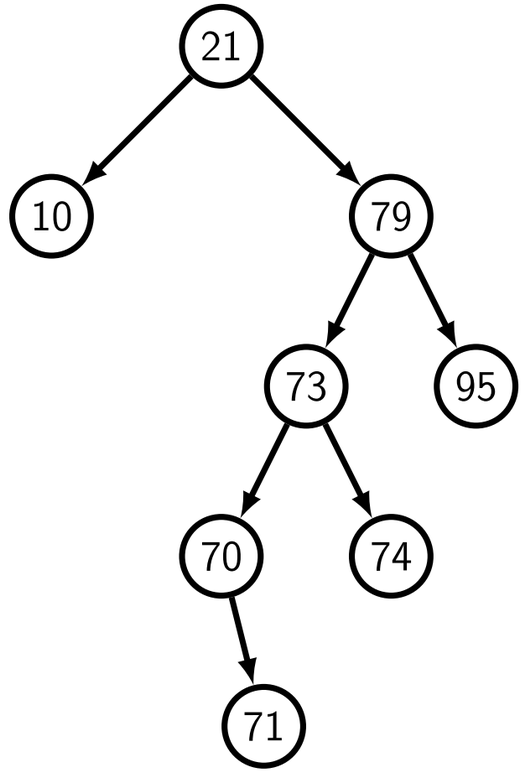
Remember, balance factor is the height of the left subtree - height of the right subtree



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

Remember, balance factor is the height of the left subtree - height of the right subtree

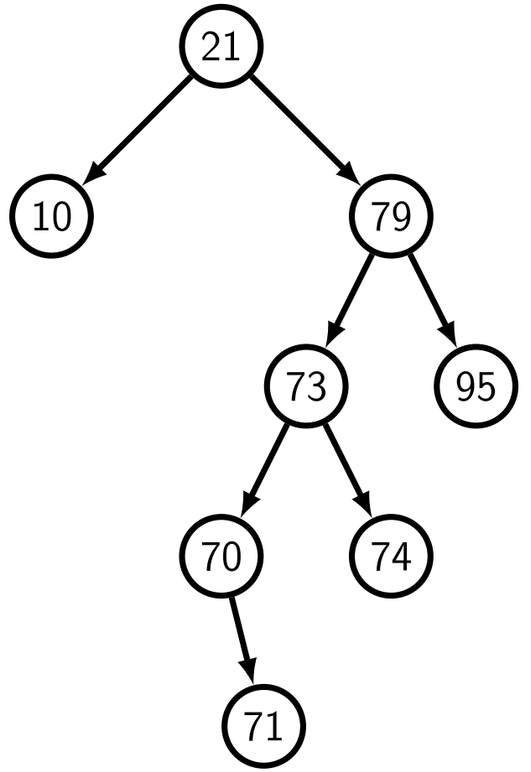
```
1  private class Node<E> {
2    private E key;
3    private Node<E> left;
4    private Node<E> right;
5    private int height;
6
7    public int getBalanceFactor() {
8
9      // height of left – height of right
10
11   }
12 }
```



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

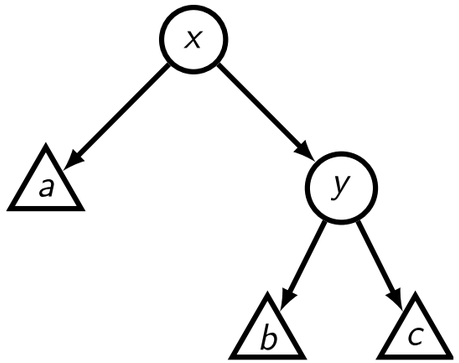
Remember, balance factor is the height of the left subtree - height of the right subtree

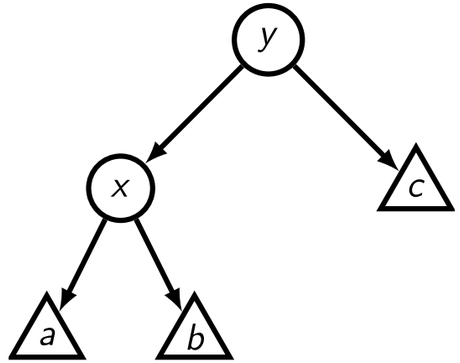
```java
private class Node<E> {
  private E key;
  private Node<E> left;
  private Node<E> right;
  private int height;

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

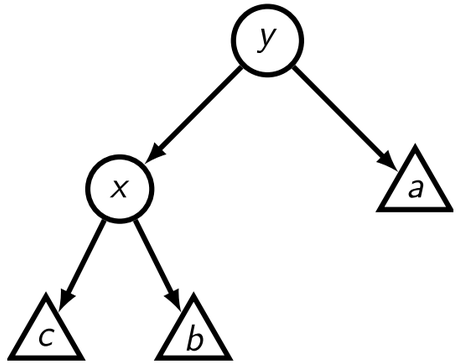


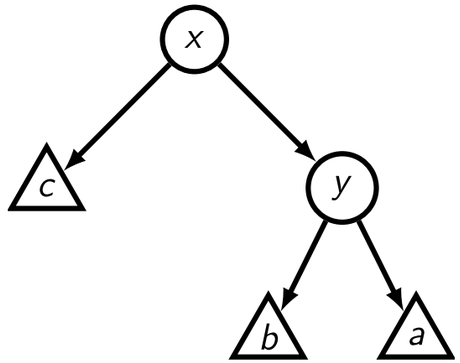
So what can we do with this balance factor? (which can be computed from the updated `height` during every `add` or `remove`).

ROTATIONS









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

1. Perform `add`/`remove` as in usual BST operations.

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

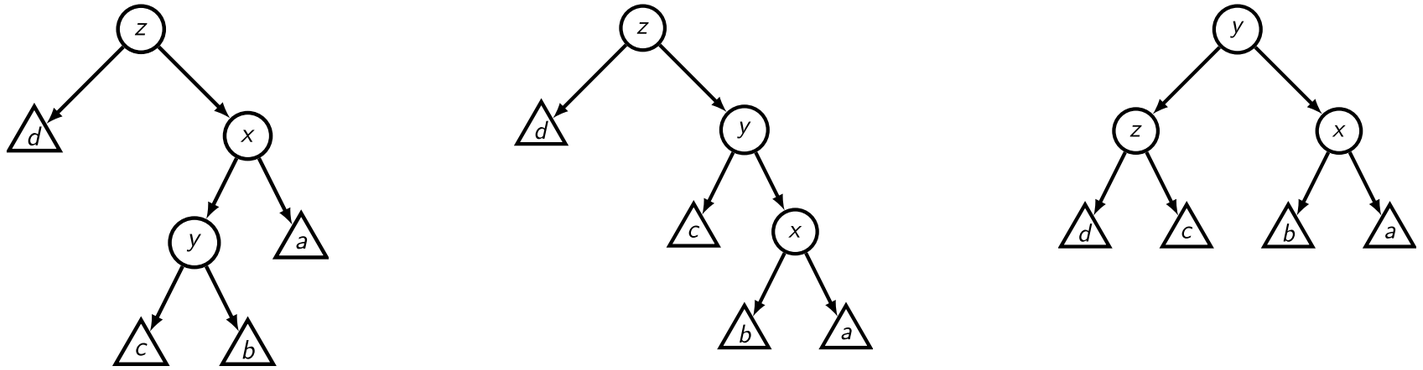
1. Perform `add`/`remove` as in usual BST operations.
2. Update node heights and calculate `node` balancing factor `bf(node)`.

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. To `balance` the node, there are 4 cases!

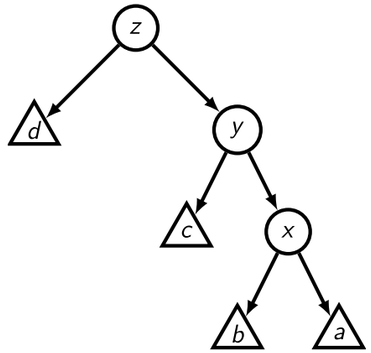
Maintaining AVL self-balancing binary search tree: Case #1

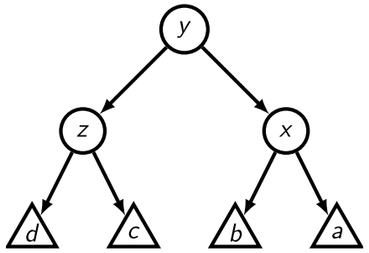
1. Perform `add`/`remove` as in usual BST operations.
2. Update node heights and calculate `node` balancing factor `bf(node)`.
3. To `balance` the node, there are 4 cases!
   Case #1: `bf(node) < -1` (tree is right-heavy) and `bf(node.right) > 0` (right tree is left-heavy)
   rotate right-left, i.e. `rotateRight(node.right)` then `rotateLeft(node)`.



Maintaining AVL self-balancing binary search tree: Case #2

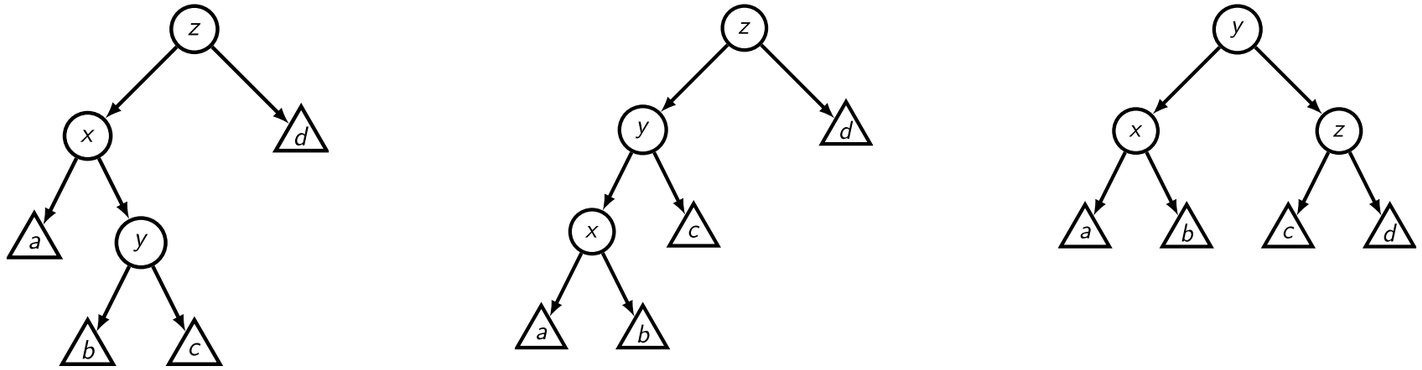
1. Perform `add`/`remove` as in usual BST operations.
2. Update node heights and calculate `node` balancing factor `bf(node)`.
3. To `balance` the node, there are 4 cases!
   Case #2: `bf(node) < -1` (tree is right-heavy) and `bf(node.right) <= 0` (right tree is right heavy)
   rotate left, i.e. `rotateLeft(node)`





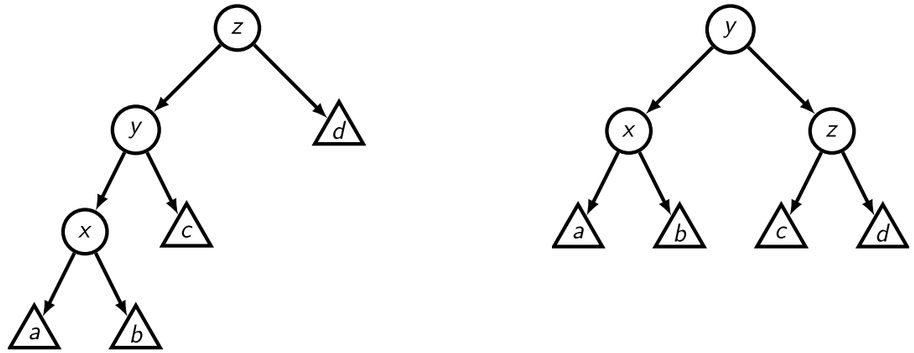
Maintaining AVL self-balancing binary search tree: Case #3

1. Perform `add`/`remove` as in usual BST operations.
2. Update node heights and calculate `node` balancing factor `bf(node)`.
3. To `balance` the node, there are 4 cases!
   Case #3: `bf(node) > 1` (tree is left-heavy) and `bf(node.left) < 0` (left tree is right-heavy)
   rotate left-right, i.e. `rotateLeft(node.left)` then `rotateRight(node)`.



Maintaining AVL self-balancing binary search tree: Case #4

1. Perform `add`/`remove` as in usual BST operations.
2. Update node heights and calculate `node` balancing factor `bf(node)`.
3. To `balance` the node, there are 4 cases!
   Case #4: `bf(node) > 1` (tree is left-heavy) and `bf(node.left) >= 0` (left tree is left-heavy)
   rotate right, i.e. `rotateRight(node)`.

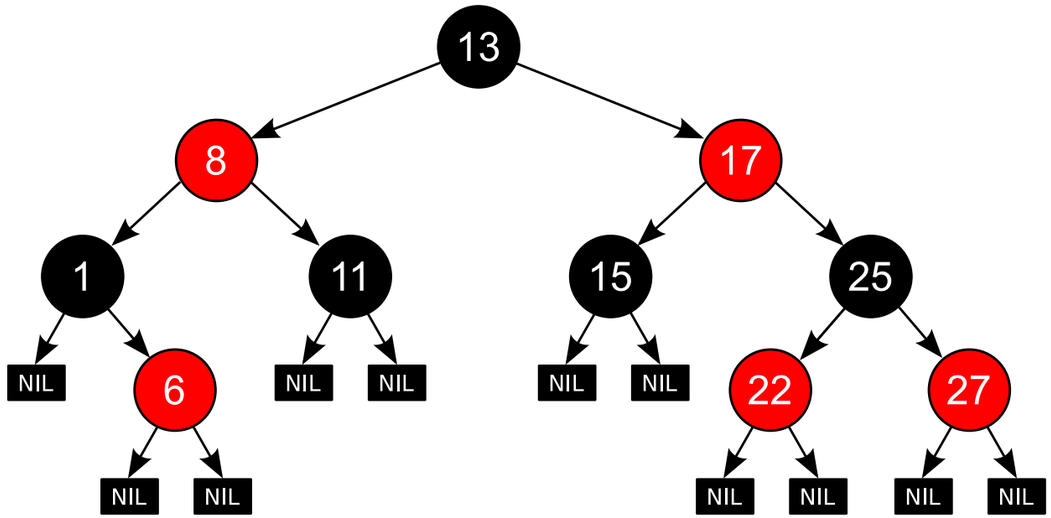


Practice: draw the AVL tree resulting from these operations.

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

Additional notes:

- Homework 7 due today at 5PM: implement a max-heap.
- There are now some additional times later in the week for the programming exam, if you'd like to move your slot.
- A study guide will be posted soon for midterm #2!
  - Expect questions on content since the first midterm (linked lists, stacks, queues, trees (including heaps and BSTs))
  - Also expect some questions on sorting
- `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)