Middlebury

CSCI 201: Data Structures

Spring 2025

Lecture 9M: Binary Search Trees

Goals for today:

- **Motivation:** how to know if a collection `contains` an item (efficiently)? How to maintain a unique, ordered set of items?
- Use a `TreeSet` to maintain an ordered set of keys.
- Use a `TreeMap` to maintain an ordered set of key-value pairs.
- Implement your own `BinarySearchTree`:
  - `add` an item to a binary search tree.
  - `remove` an item from a binary search tree.
  - Check if a BST `contains` a key.
  - Use a BST to `get` the value associated with a key.

Motivation: does a collection `contains` an item?

- `ArrayList`: 

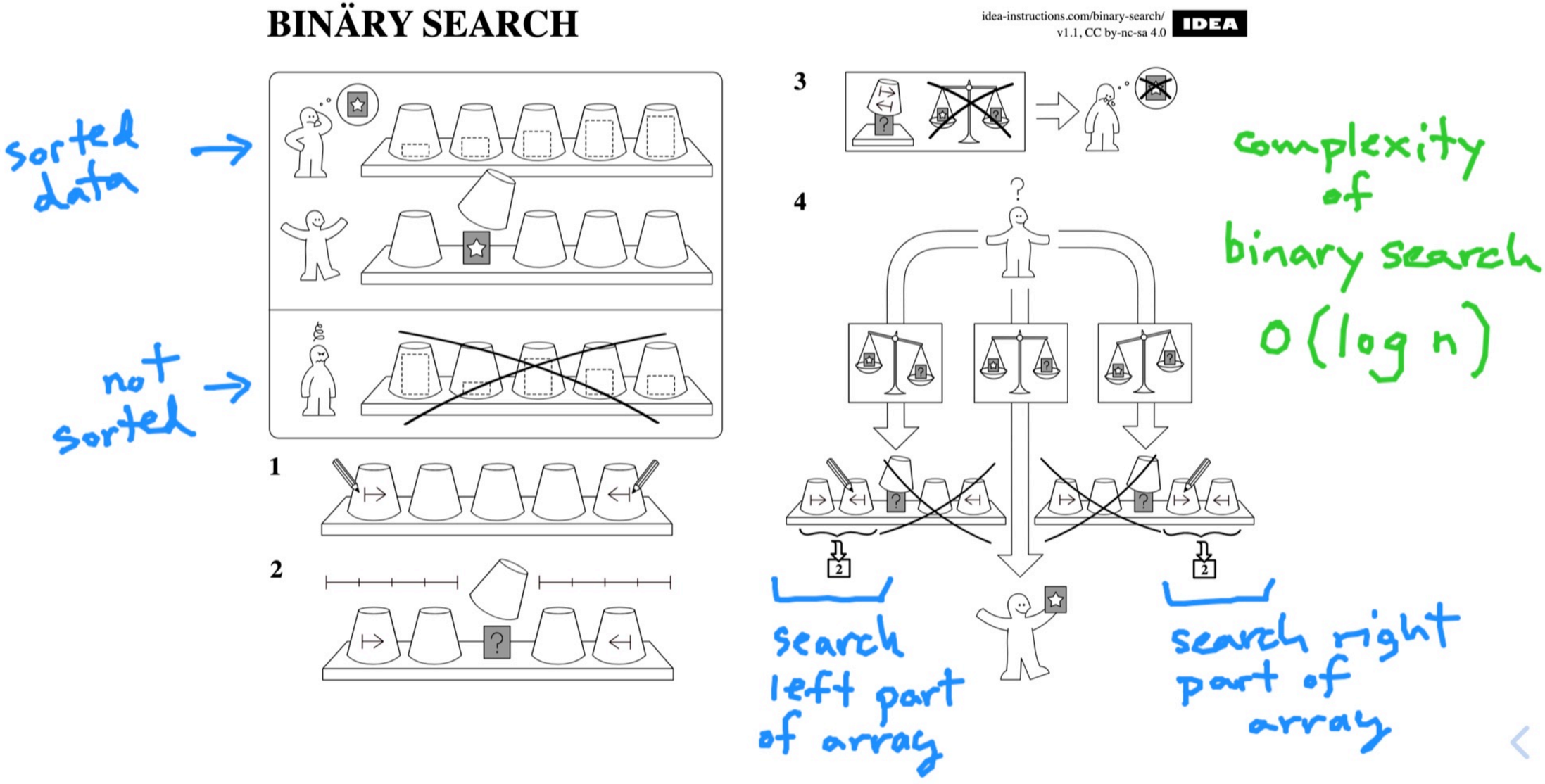
O(n)

- `LinkedList`:

O(n)

💡 **Idea:** maintain sorted (ordered) items, perform binary search.

Inspiration: binary search to see if a sorted array has an item.



Binary search on our favorite data structures?

With the requirement that we can `add` or `remove` items.

- `ArrayList`: 

add need to shift data

↳ O(n)

- `LinkedList`:

add need to traverse list

↳ O(n)

💡 **Idea:** linked structure, maintain order between nodes.

How Java does it: trees!

Main things we want:

- Determine if this key exists in the BST (`contains`, `containsKey`).
- Ability to `add` (or `put`) a new item into a tree.
- Ability to `remove` an item from the tree.

```java
import java.util.*;

public class TreeSetMapExamples {

  public static void main(String[] args) {
    TreeSet<Integer> set = new TreeSet<>();

    set.add(12);
    set.add(8);
    set.add(14);
    set.add(5);
    set.add(9);
    set.add(20);

    System.out.println(set);
    for (int item : set) {
      System.out.println(item);
    }
    System.out.println(set.contains(12));
    System.out.println(set.contains(13));
  }
}
```

```java
import java.util.*;

public class TreeSetMapExamples {

  public static void main(String[] args) {
    TreeMap<Character, Integer> map
                                = new TreeMap<>();
    String phrase = "bananabread";
    for (int i = 0; i < phrase.length(); i++) {
      Character c = phrase.charAt(i);
      if (map.containsKey(c)) {
        map.put(c, map.get(c) + 1);
      } else {
        map.put(c, 1);
      }
    }
    System.out.println(map);
    System.out.println(map.containsKey('b'));
    System.out.println(map.containsKey('f'));
  }
}
```

Binary Search Trees (BST): maintain order between left and right child nodes (subtrees) in relation to each node.

Main things we want:

- Determine if this key exists in the BST (`contains`, `containsKey`).
- Ability to `add` (or `put`) a new item into a tree.
- Ability to `remove` an item from the tree.

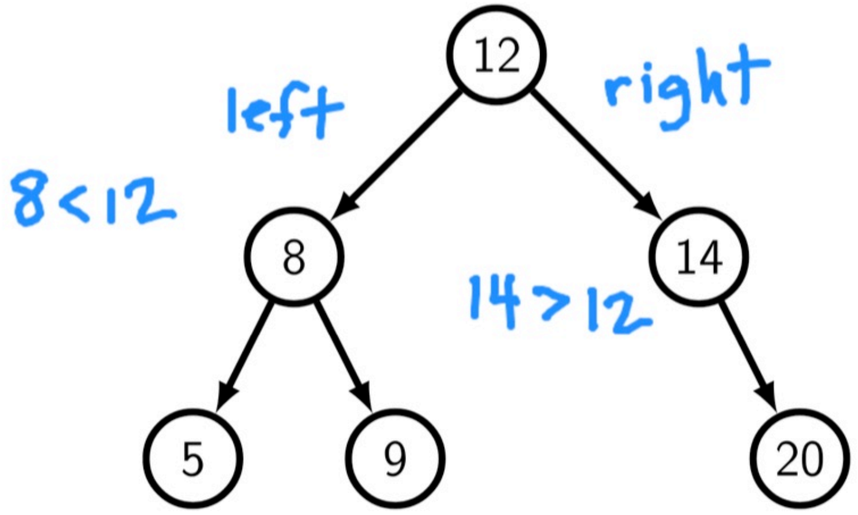
We need to be able to compare keys

<, >, ==

compareTo

Main idea:

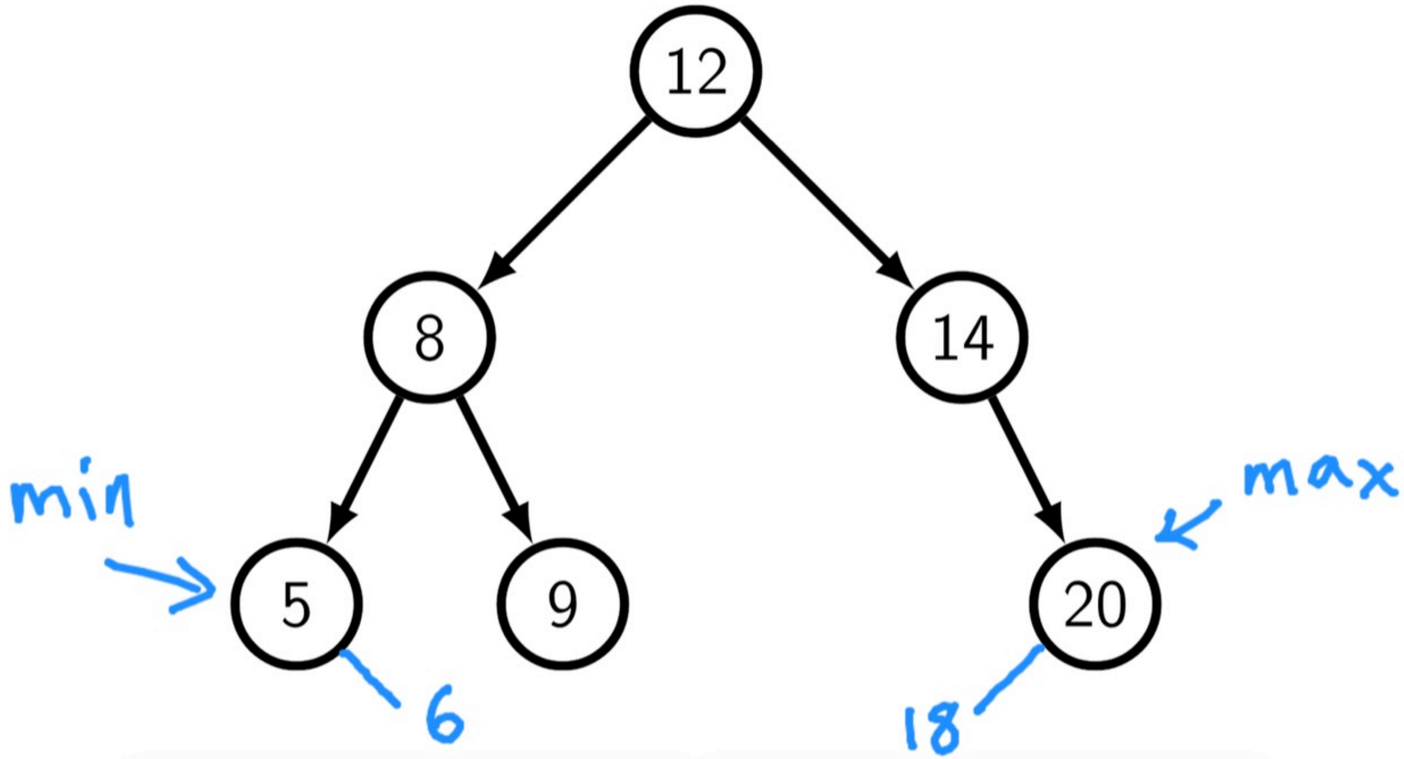
- All items (keys) in left subtree are $<$ item (key) at current node;
- All items (keys) in right subtree are $>$ item (key) at current node.




```java
class BSTSet<E extends Comparable<E>> {
  class Node {
    public E key;
    public Node left;
    public Node right;
  }
}
```

```java
class BSTMap<K extends Comparable<K>, V> {
  class Node {
    public K key;
    public V value;
    public Node left;
    public Node right;
  }
}
```

Retrieving the minimum & maximum item/key in a BST.




```java
public E getMin() {
  Node node = root;
  while (node.left != null) {
    node = node.left;
  }
  return node.key;
}
```

```java
public E getMax() {
  Node node = root;
  while (node.right != null) {
    node = node.right;
  }
  return node.key;
}
```

See Slido # 3590710

CS 201 Lecture 16

Which of the following would process the items (keys) from a BST in sorted order (smallest to largest)? 30

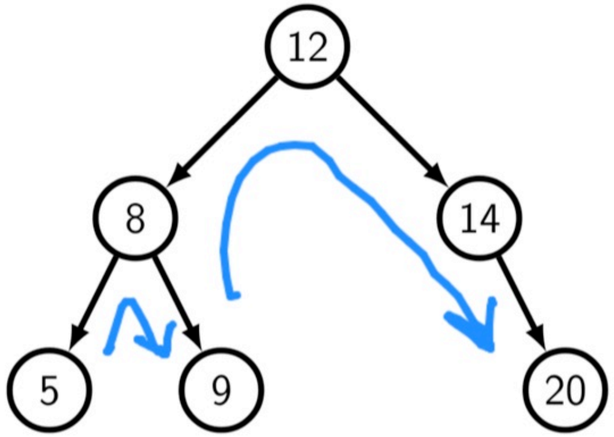
- ○ Pre-order traversal
- ● In-order traversal
- ○ Post-order traversal
- ○ Level-order traversal

Send

Voting as Anonymous

slido

Acceptable Use - Slido Privacy - Cookie Settings

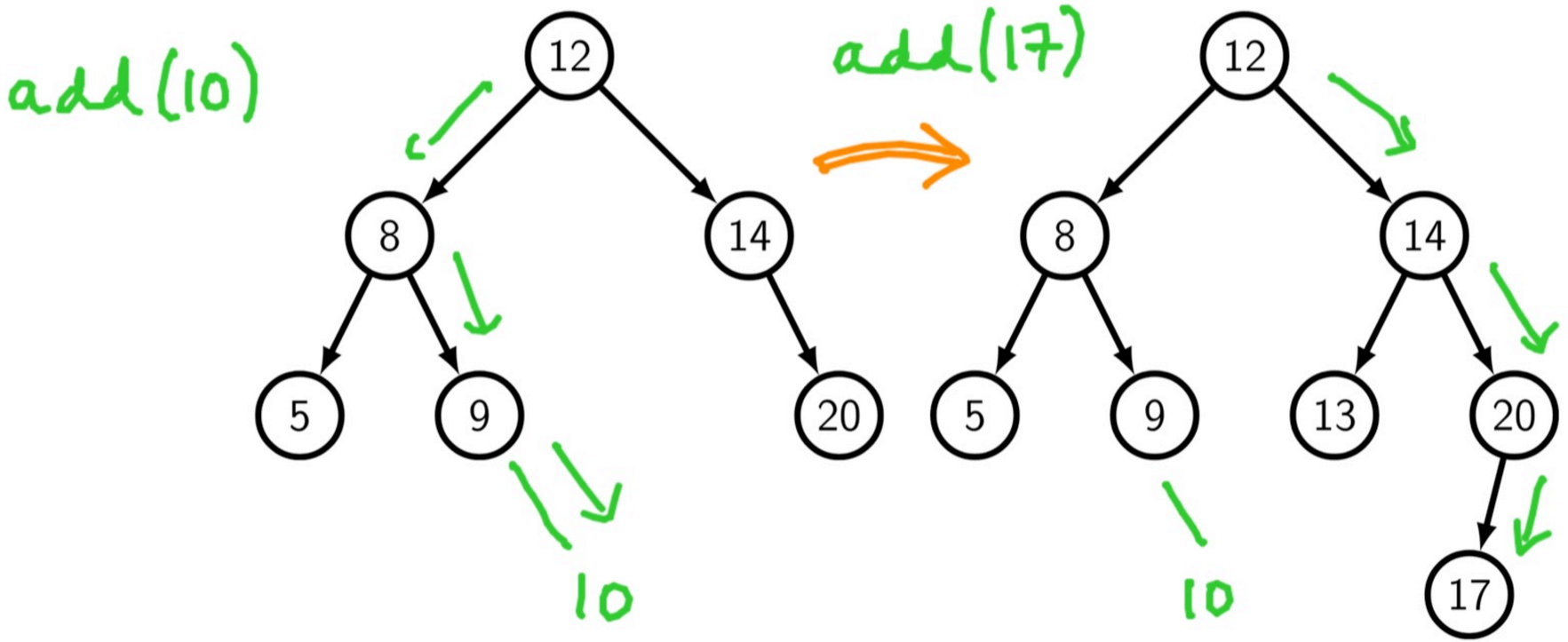


In-order:

- visit left subtree (recursively)
- visit root
- visit right subtree (recursively)

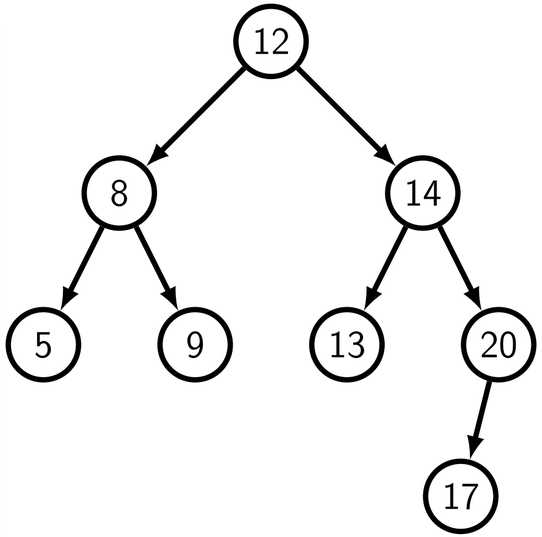
Adding a key to a BST (add, put).

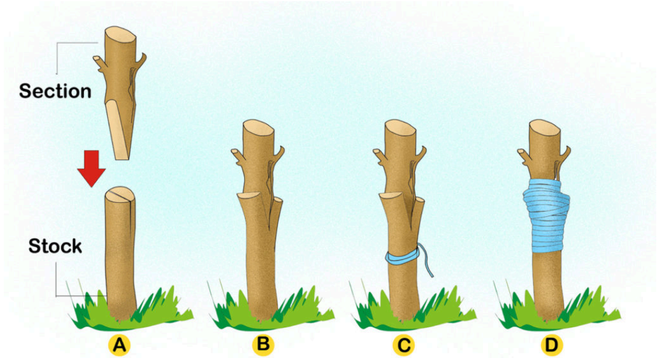
- Start at `node = root`.
- If `key < node.key`, need to put in left subtree (`node.left`).
- If `key > node.key`, need to put in right subtree (`node.right`).
- If `key == node.key`, key is already in tree! Done.
- If `node == null`, create a new `Node` to hold this `key`.



Implementing the add method.

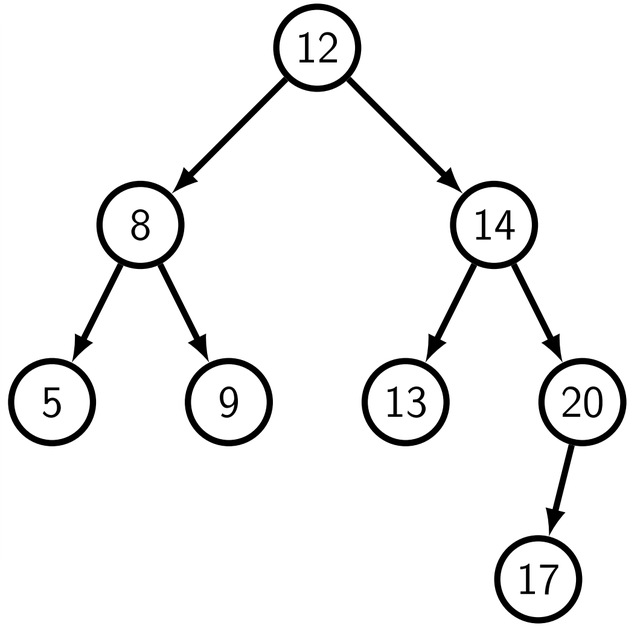
```java
public void add(E key) {
  root = add(root, key);
}

private Node add(Node node, E key) {
  if (node == null) {
    // last node was a leaf, create the new leaf node
    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);
  }
  return node;
}
```





Exercise: implement the `contains` method and test it!

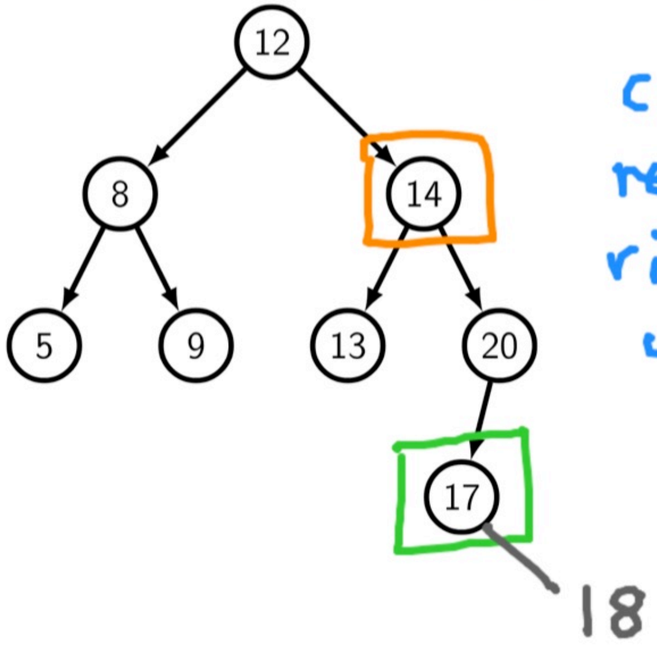
```java
public boolean contains(E key) {
    return contains(root, key);
  }

  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);
    }
  }
```



Removing an item/key from a BST:

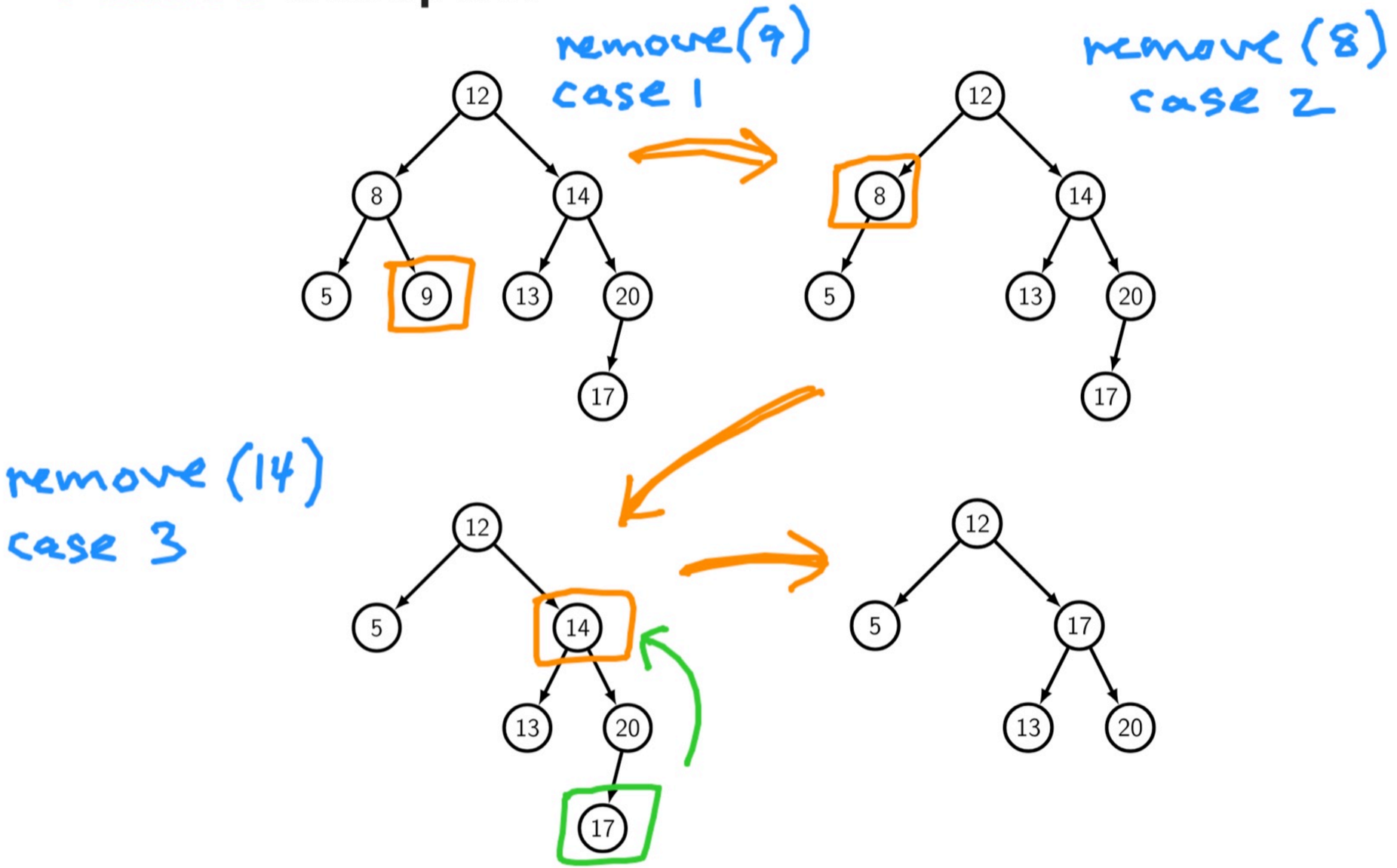
- Start at `node = root`.
- If `key < node.key`, need to remove node in left subtree (`node.left`).
- If `key > node.key`, need to remove node in right subtree (`node.right`).
- If `key == node.key`, three cases to consider:
  1. Is this a leaf? Delete node.
  2. (a) If `node.left == null`, "graft" `node.right`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

eg, remove (14)



case 3 removal of min in right subtree will be a case 1 or case 2 removal

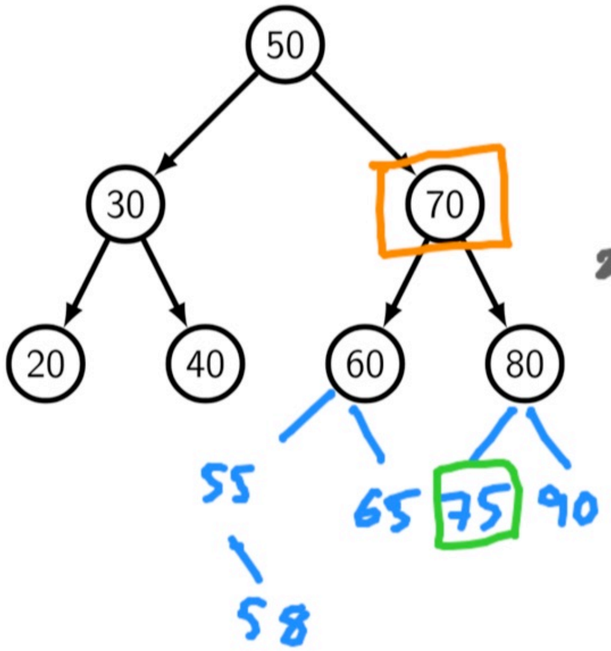
remove examples.

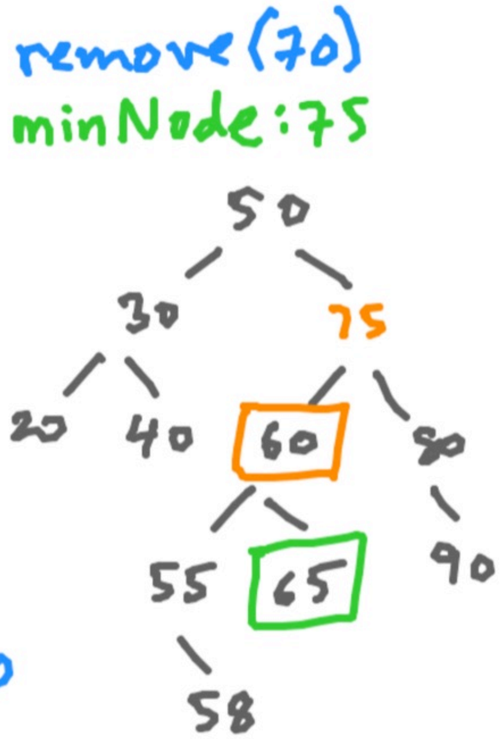


Exercise: draw the tree after the following calls to `add` and `remove`.

Work on the whiteboards in groups.

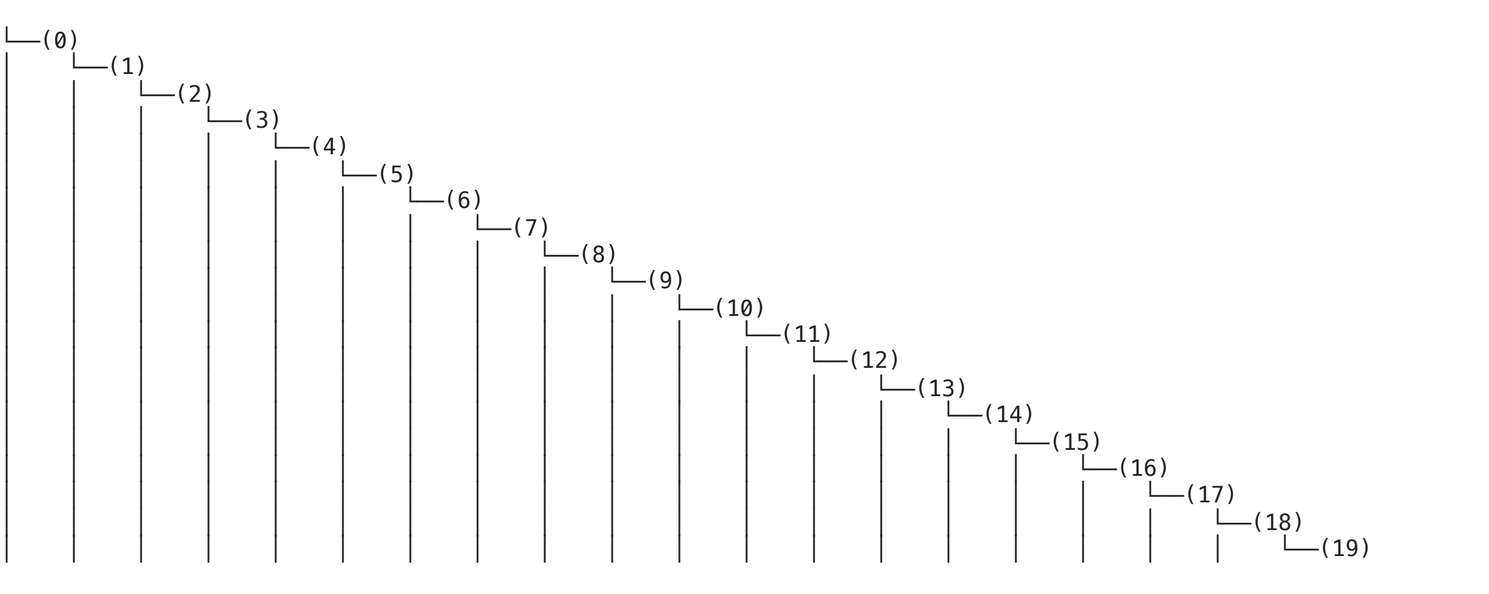
- `add`: 55, 65, 58, 75, 90
- then `remove`: 70, 60, 50





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);
```



Additional notes:

- Techniques to *balance* trees next class!
- Homework 7 due on Thursday 4/17: implement a max-heap.
- There is also an optional Homework 7 challenge problem.
- See Oracle's documentation for `TreeSet`: https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html
- See Oracle's documentation for `TreeMap`: https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html