Lecture 22: Binary Search Trees

CS 62

Fall 2018

Alexandra Papoutsaki & William Devanny

```
public class Student {
        private String name;
        private int age;

        public Student(String name, int age) {
                this.name = name;
                this.age = age;
        }
        public String getName() {
                return name;
        }
         public int getAge() {
                return age;
        }
        @Override
        public String toString() {
                return "name=" + this.getName()+" age="+this.getAge();
         }
}
```

How can you compare students by name in ascending order?

How can you compare students by age in descending order?

```
public class NameComparator implements Comparator<Student> {
      @Override
      public int compare(Student o1, Student o2) {
            return o1.getName().compareTo(o2.getName());
       }
}

public class AgeComparator implements Comparator<Student> {
      @Override
      public int compare(Student o1, Student o2) {
            return o2.getAge() - o1.getAge();
       }
}
```

Binary Search Trees

A binary tree is a binary search tree iff

- it is empty or
- if the value of every node is both greater than or equal to every value in its left subtree and less than or equal to every value in its right subtree.

- Definition allows duplicate values of the root node to fall on either side, but our code will prefer to have them on the left.

BST ADT

```
public class BinarySearchTree<E extends Comparable<E>> {
      protected BinaryTree<E> root;
      protected Comparator<E> ordering;
      public BinarySearchTree();
      public BinarySearchTree(Comparator<E> alternateOrder);
      public void add(E value);
      public boolean contains(E value);
      public E remove(E value);
      protected BinaryTree<E> locate(BinaryTree<E> root, E val);
      protected BinaryTree<E> predecessor(BinaryTree<E> node);
      protected Iterator<E> iterator(); //in-order traversal
}
```

Locating a Value

- Useful for `add`, `contains`, and `remove`
- Returns a pointer to the node with a given value
  - …or to a node where that exact value could be added
- Recursive implementation (could be iterative)

Locating a Value

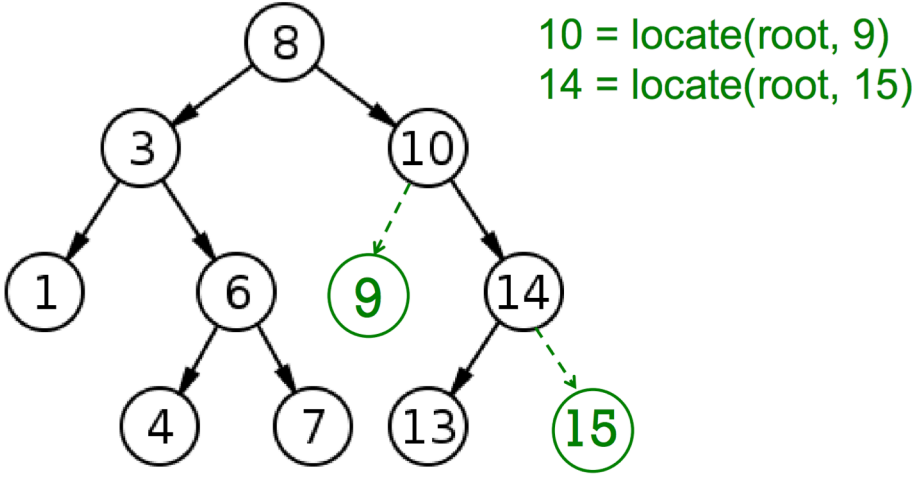
- Check current value vs. the search value
  - If equal, return this node
  - If smaller, locate within left subtree
    - Else within right subtree
  - If the appropriate subtree is empty, return this node

```
protected BinaryTree<E> locate(BinaryTree<E> root, E value){
    E rootValue = root.value();
    BinaryTree<E> child;
    if (rootValue.equals(value)) return root; // found at root
    // look left if less-than, right if greater-than
    if (ordering.compare(rootValue,value) < 0) {
        child = root.right();
    } else {
        child = root.left();
    }
    // no child there: not in tree, return this node,
    // else keep searching
    if (child.isEmpty()) {
        return root;
    } else {
        return locate(child, value);
    }
}
```

Using `locate` to add a node

Case One: `locate` returns pointer to where node should be added

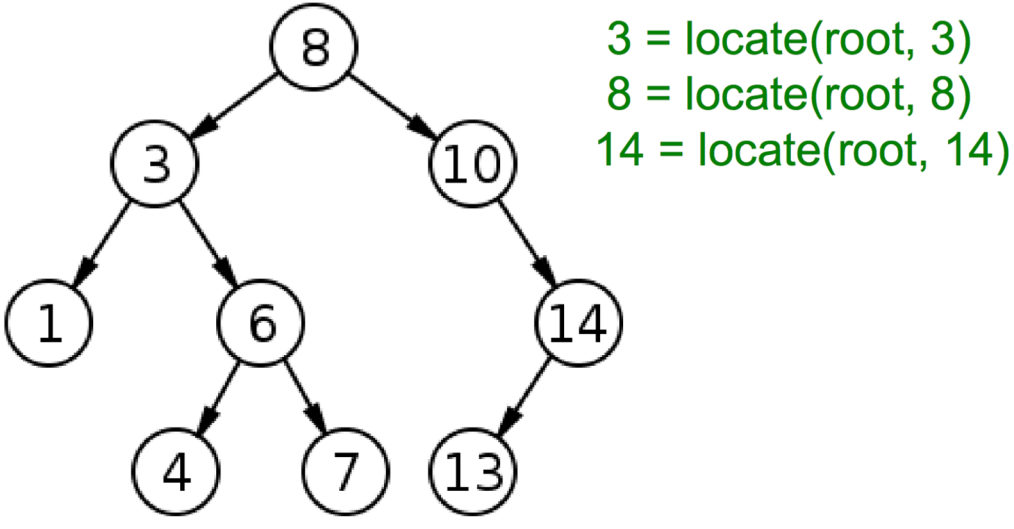
- If value less than returned node, create new left child
- If value greater than returned node, create new right child



Using `locate` to add a node

Case Two: `locate` returns pointer to node with same value

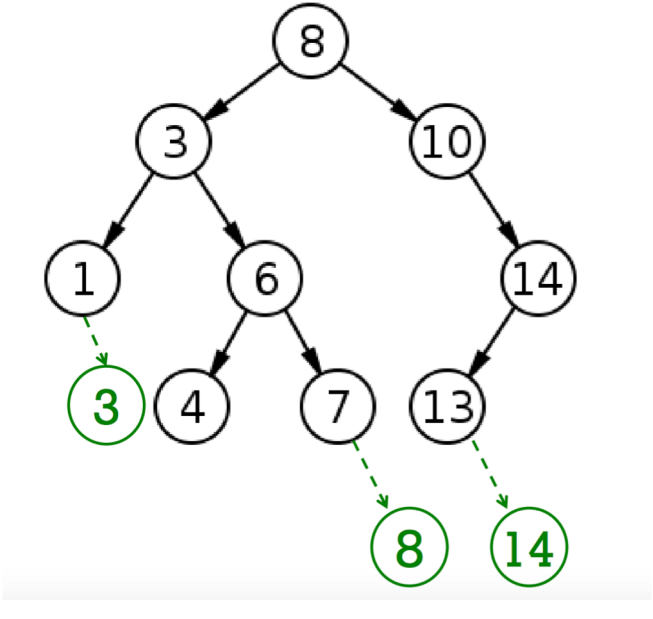
- Duplicates go in left subtree (could have chosen right)
- Where in the left subtree?



Using `locate` to add a node

Case Two: `locate` returns pointer to node with same value

- Duplicates go in left subtree (could have chosen right)
- *Should be the rightmost descendant of left tree*



Predecessor and Successor

Predecessor: the rightmost descendent in left subtree

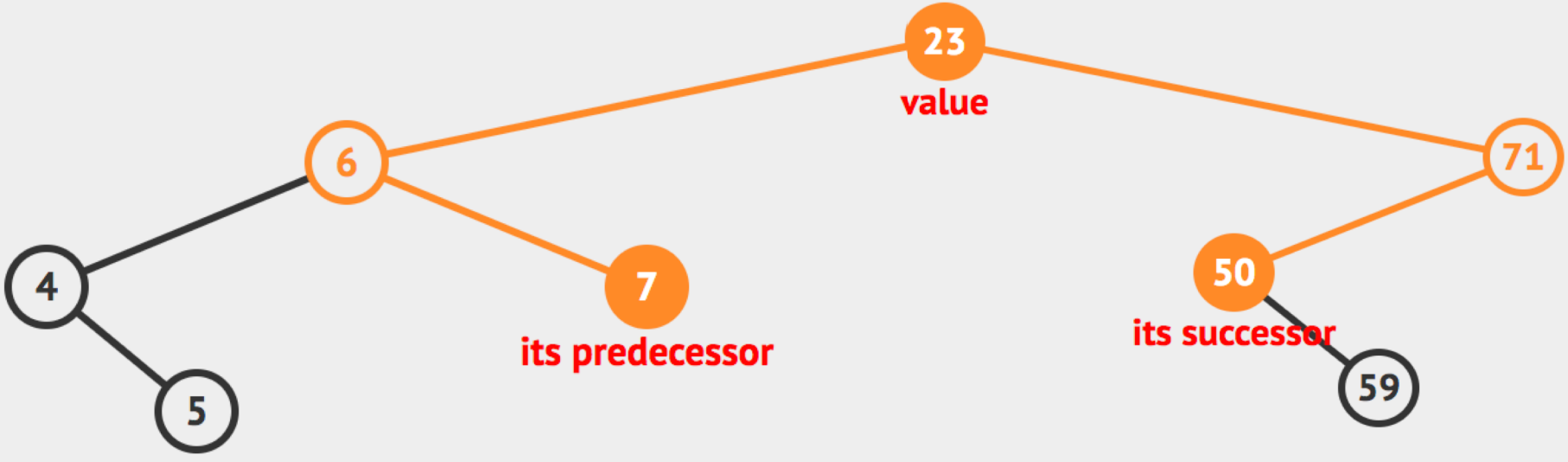
- The next-smaller value in the tree

Successor: the leftmost descendent in right subtree

- The next-larger value in the tree

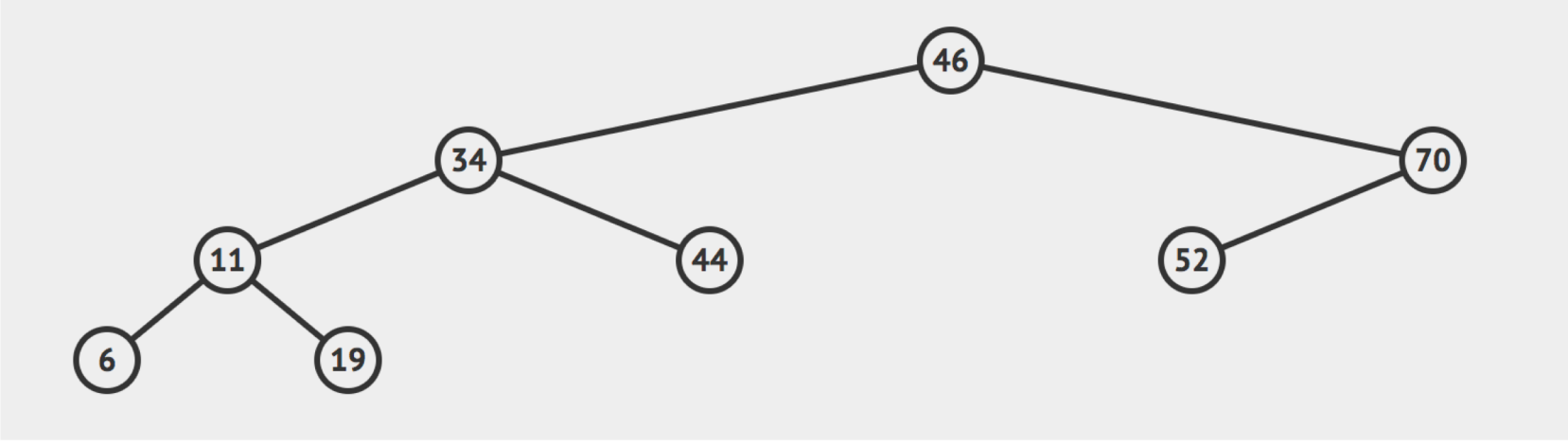
```
protected BinaryTree<E> predecessor(BinaryTree<E> root) {
        BinaryTree<E> result = root.left();
        while (!result.right().isEmpty()) {
            result = result.right();
        }
        return result;
    }

    protected BinaryTree<E> successor(BinaryTree<E> root) {
        BinaryTree<E> result = root.right();
        while (!result.left().isEmpty()) {
            result = result.left();
        }
        return result;
    }
```

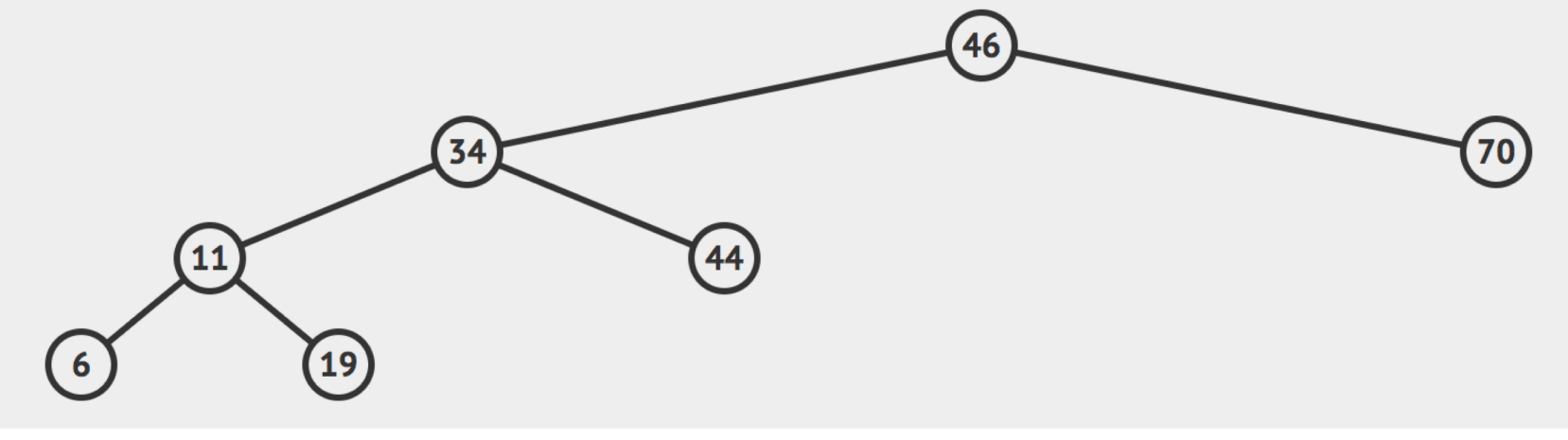

23
value
6
71
4
7
50
its predecessor
its successor
5
59

```
public void add(E value) {
     BinaryTree<E> newNode = new BinaryTree<E>(value,EMPTY,EMPTY);
     // add value to binary search tree
     // if there's no root, create value at root
     if (root.isEmpty()) {
         root = newNode;
     } else {
         BinaryTree<E> insertLocation = locate(root,value);
         E nodeValue = insertLocation.value();
         // The location returned is the successor or predecessor
         // of the to-be-inserted value
         if (ordering.compare(nodeValue,value) < 0) {
             insertLocation.setRight(newNode);
         } else {
             if (!insertLocation.left().isEmpty()) {
                 // if value is in tree, we insert just before
                 predecessor(insertLocation).setRight(newNode);
             } else {
                 insertLocation.setLeft(newNode);
             }
         }
     }
     count++;
 }
```

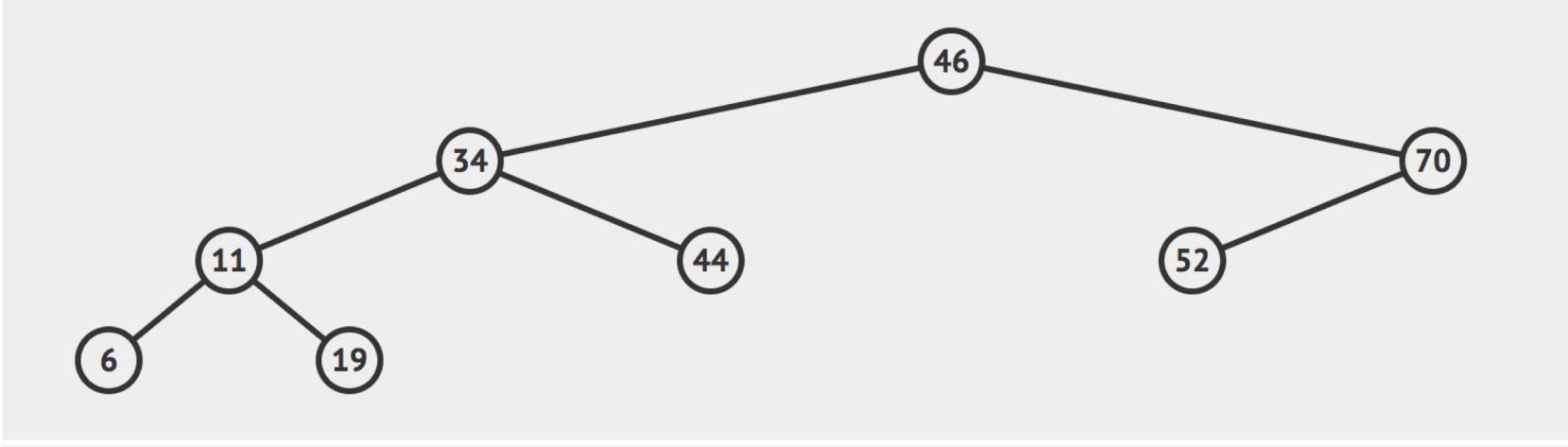
Removing nodes – Node is a leaf



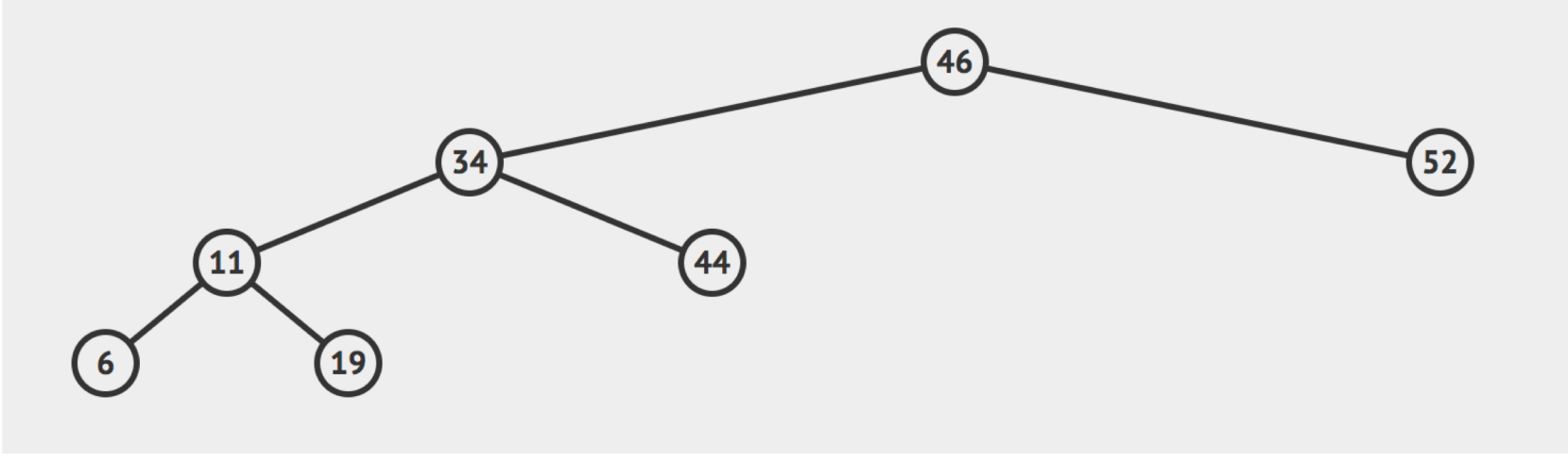
remove(52)



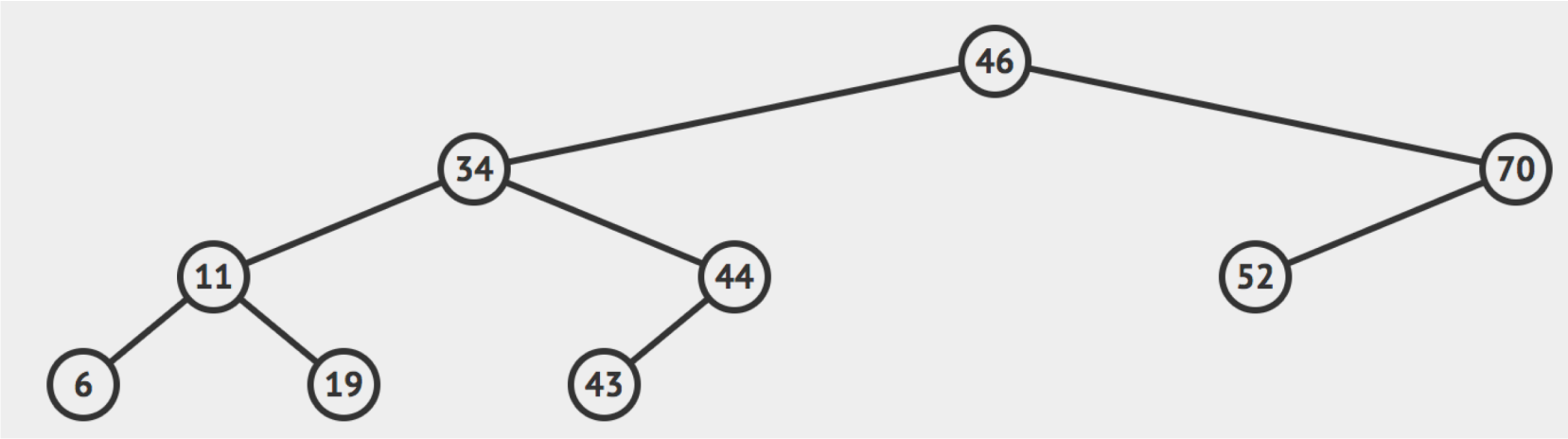
Removing nodes – only one child



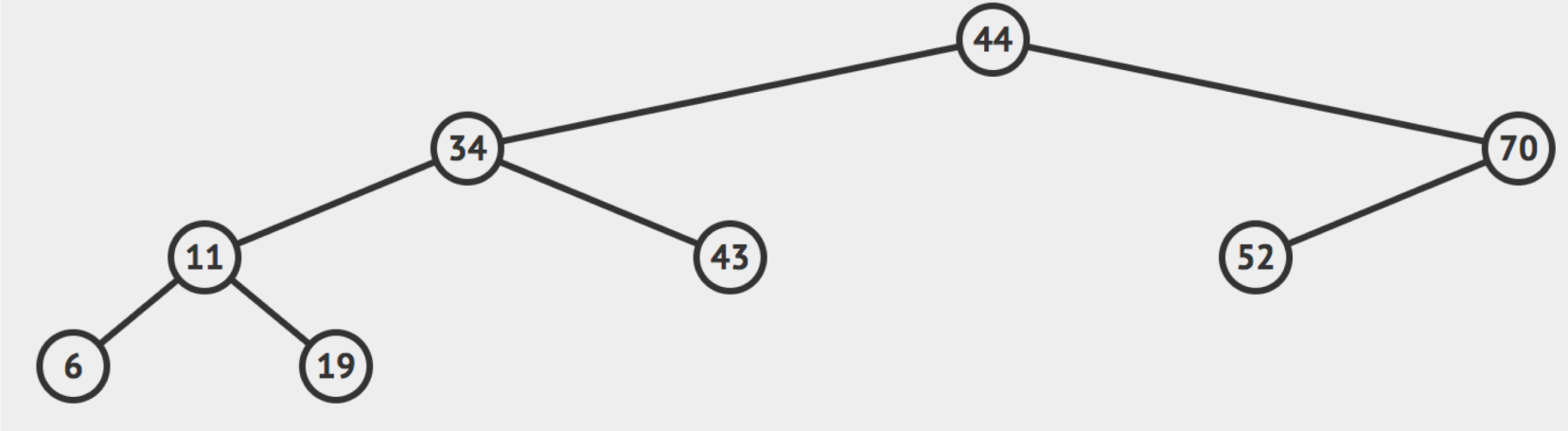
remove(70)



Removing a node with two subtrees



Remove the node and substitute with predecessor



remove(46)

Removing nodes

- Calling `remove(E val)` removes node with value `val`
- Predecessor of root becomes new root
  - Predecessor is in left subtree
  - Predecessor has no right subtree

  - Complexity is $O(h)$ where h is height of tree
    - Worst-case $O(h)$ to locate
    - Worst-case $O(h)$ to find predecessor

Complexity

- `locate`, `add`, `contains`, `remove` are all $O(h)$
- Can we guarantee that h is $O(\log n)$?
  - Only if tree stays balanced!!
- Binary search trees that stay balanced:
  - AVL trees
  - Red-black trees
- We'll do splay trees, which don't guarantee balance
  - but guarantee good average behavior
  - easier to understand than alternatives
  - better than others if likely to go back to recent nodes