Lecture 17: Comparable, Comparator, and Binary Search Trees

CS 2110, Matt Eichhorn and Leah Perlmutter

October 23, 2025

Announcements

- TA evaluations due Friday at 5pm
  - https://apps.engineering.cornell.edu/TAEval/survey.cfm --->
  - if you're not sure your TA's name for the eval, check the staff roster for their photo
- A8 released
  - long, hard, lots of reading (both code and English)
  - start early! go to office hours!
- Know your support resources
  - course website → About → Tips for Success

Today's Learning Outcomes

Tree Data Structures

1. Describe the binary search tree invariant and determine whether it is satisfied by a given tree.
2. Write modifying methods on a binary search tree that preserve its invariant.
3. Analyze the time/space complexities of a given method on a binary search tree.

Comparator

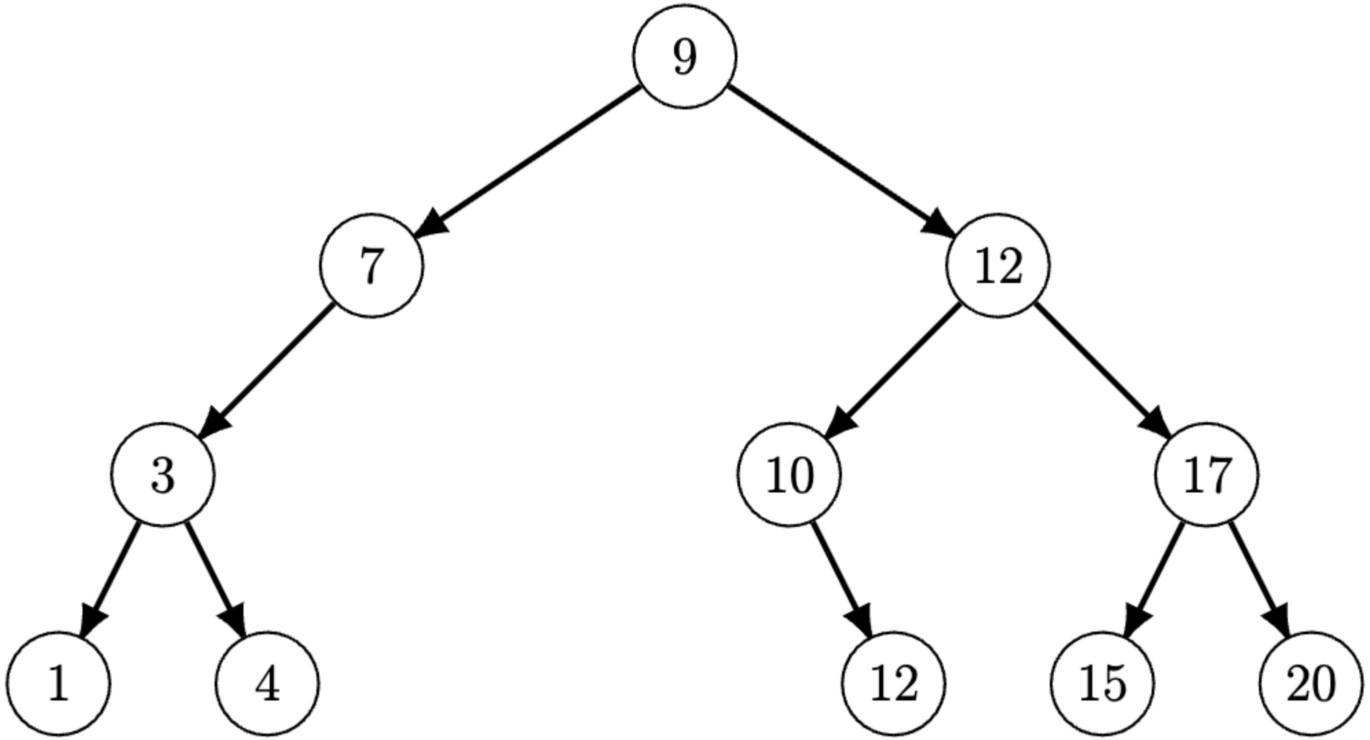
1. Describe the use cases for the Comparable and Comparator interfaces.
2. Explain the semantics of generic type bounds and write code that incorporates them.

Sit near a few people

- This is a high interaction day! I’ll ask you to discuss several questions with a partner or trio.
-

Binary Search Trees

Review: In-order Tree Traversal

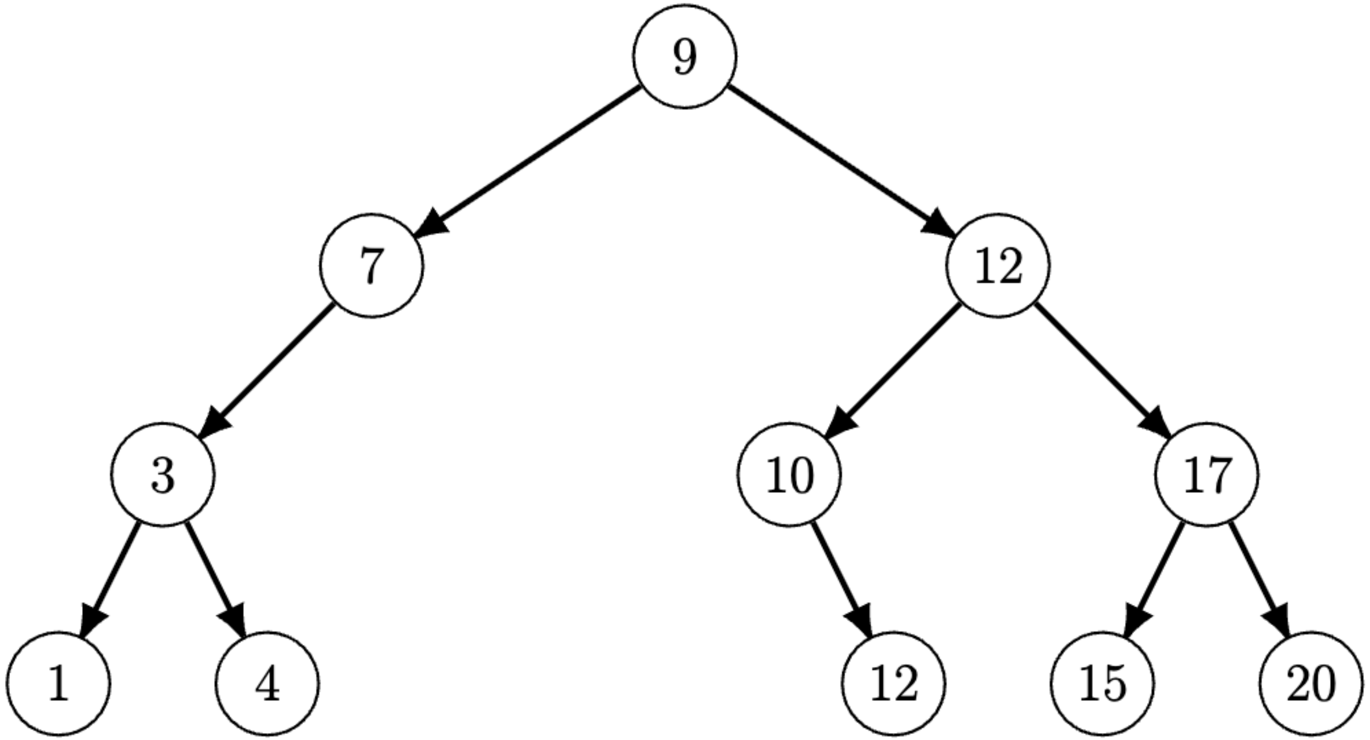


Write an *in-order* traversal of this tree.

(answer on next slide)

PollEv.com/leahp
text **`leahp`** to **`22333`**

Review: In-order Tree Traversal

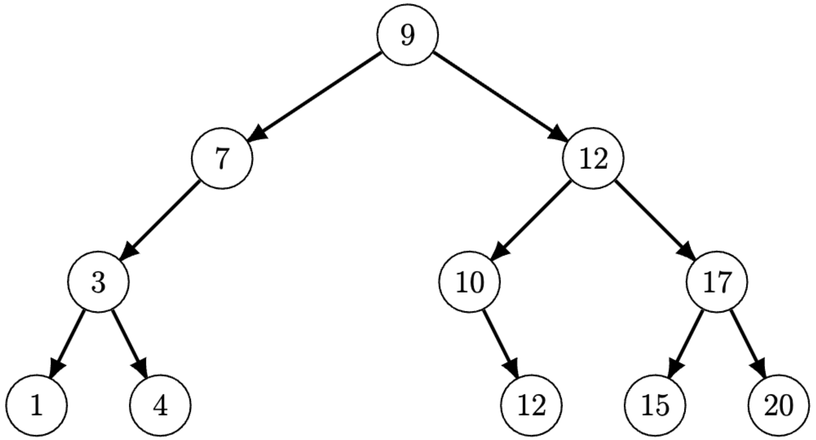


Write an *in-order* traversal of this tree.
1, 3, 4, 7, 9 , 10, 12, 12, 15, 17, 20

PollEv.com/leahp
text **`leahp`** to **`22333`**

Binary Search Trees (BSTs)

- BST Order Invariant: the nodes of the tree appear in sorted order from least to most in an in-order traversal
- It is easy to perform binary search on the tree (see animation in lecture notes)
- How can we maintain a tree with the BST Order Invariant when adding new nodes?



Interlude:
Comparing Objects

Comparable Interface

```java
int compareTo(T other);
```

- Return positive if `this` > `other`
- Return negative if `this` < `other`
- Return 0 if `this` is equal to `other`

Properties of compareTo()

- Switching order flips sign
  - positive `x.compareTo(y)` ⇔ negative `y.compareTo(x)`
- Transitivity
  - `x.compareTo(y)>0` AND `y.compareTo(z)>0` → `x.compareTo(z)>0`
- Equivalent object behavior (required)
  - if `x.compareTo(y)== 0`
    - then sign of `x.compareTo(z)` equals sign of `y.compareTo(z)`
- Consistency with equals (desirable)
  - `x.compareTo(y) == 0` exactly when `x.equals(y)`

Comparable Point

```java
/** An immutable point in the 2D coordinate plane with `double` coordinates. */
public record Point(double x, double y) implements Comparable<Point> {

  /**
   * Compares `this` and `other` based on their distance from the origin, returning
   * a positive integer when `this` is farther from the origin, a negative integer
   * when `this` is closer to the origin, and 0 when `this` and `other` are
   * equidistant from the origin.
   */
  @Override
  public int compareTo(Point other) {
    double thisDistSquared = this.x * this.x + this.y * this.y;
    double otherDistSquared = other.x * other.x + other.y * other.y;
    return (int) Math.signum(thisDistSquared - otherDistSquared);
  }

  // ... additional methods
}
```

Comparator Interface

```
int compare(T o1, T o2);
```

- Return positive if `o1` > `o2`
- Return negative if `o1` < `o2`
- Return 0 if `o1` is equal to `o2`

Requires same consistency properties as `compareTo()`

Comparator for Point

```java
/** A `Comparator` that models a lexicographic ordering of `Point`s. */
public class PointLex implements Comparator<Point> {
  /**
   * Compares `o1` and `o2` lexicographically, returning a negative integer
   * when `o1` is lexicographically earlier, a positive integer when
   * `o2` is lexicographically earlier, and 0 when `o1` and `o2` are equal.
   */
  @Override
  public int compare(Point o1, Point o2) {
      int dx = (int) Math.signum(o1.x() - o2.x());
      return dx == 0 ? (int) Math.signum(o1.y() - o2.y()) : dx;
  }
}
```

Comparable vs. Comparator

Comparable

- Class makes its own instances comparable to each other
- …

Comparator

- External class with a method that can compare objects
- …

Comparable vs. Comparator

Comparable

- Class makes its own instances comparable to each other
- Can use when you are the implementer

Comparator

- External class with a method that can compare objects
- Can use when you are not the implementer
- **Supports multiple kinds of comparison for the same type**

Multiple ways of comparing

Possible Book fields

- Author(s)
- Title
- Dewey Decimal (DDC) Number
- Library of Congress(LDC) Number
- Korean Decimal Classification (KDC) Number

Book class might have fields

- AuthorComarator
- TitleComparator
- DDCComparator
- LCCComparator
- KDCComparator

Generic Methods and Type Bounds

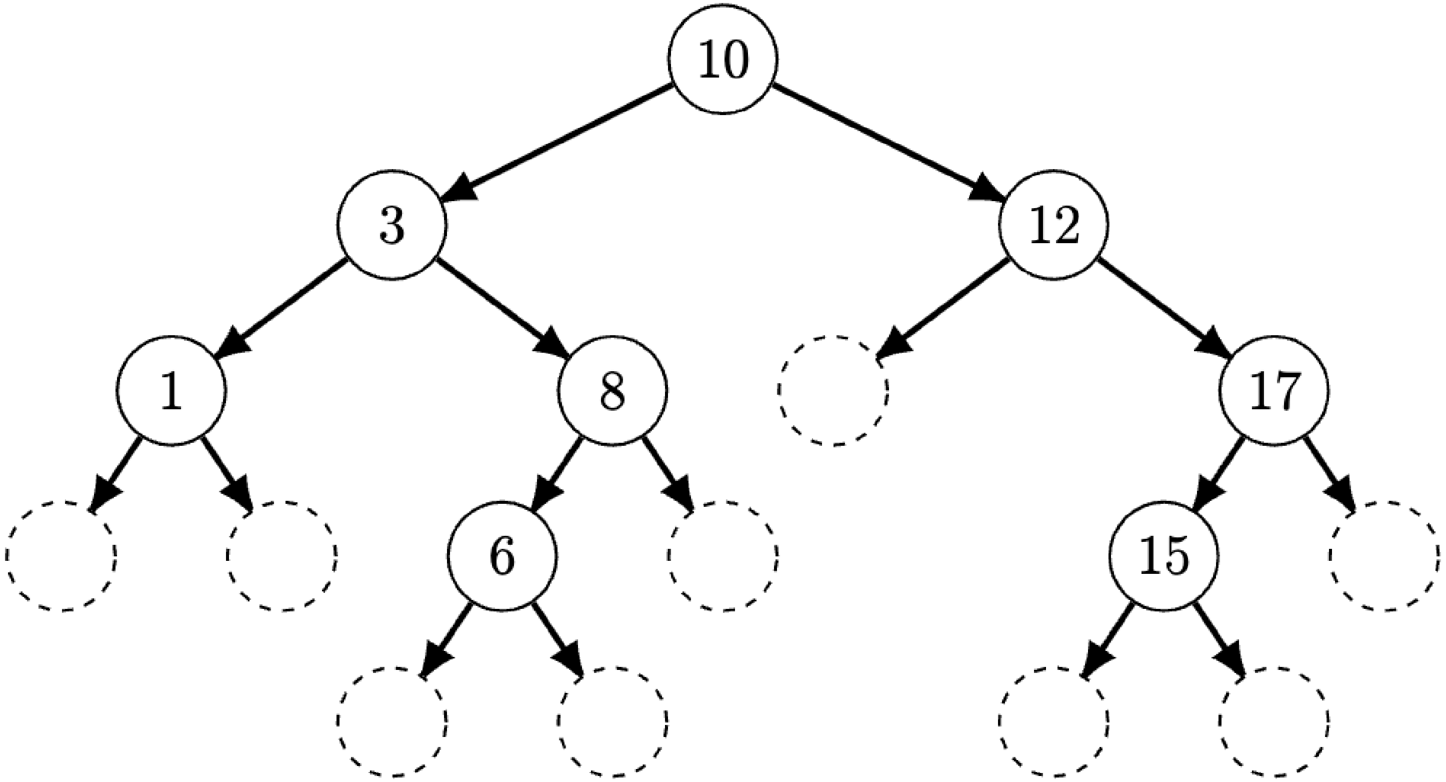
```java
/** Sorts the entries of `a` using the insertion sort algorithm. */
static <T extends Comparable<T>> void insertionSort(T[] a) {...}


/** Inserts entry `a[i]` into its sorted position in `a[..i)` such that `a[..i]` contains the
 * same entries in sorted order. Requires that `0 <= i < a.length` and `a[..i)` is sorted. */
static <T extends Comparable<T>> void insert(T[] a, int i) {...}


/**  Swaps entries `a[i]` and `a[j]`. Requires that `0 <= i < a.length` and `0 <= j <
  a.length`.*/
static <T> void swap(T[] a, int i, int j) {...}
```

Generic Insertion Sort with Comparable

```java
/** Sorts the entries of `a` using the insertion sort algorithm. */
static <T extends Comparable<T>> void insertionSort(T[] a) {
  /* Loop invariant: a[..i) is sorted, a[i..] are unchanged. */
  for (int i = 0; i < a.length; i++) {
    insert(a, i);
  }
}
/** Inserts entry `a[i]` into its sorted position in `a[..i)` such that `a[..i]` contains the
 * same entries in sorted order. Requires that `0 <= i < a.length` and `a[..i)` is sorted. */
static <T extends Comparable<T>> void insert(T[] a, int i) {
  assert 0 <= i && i < a.length; // defensive programming
  int j = i;
  while (j > 0 && a[j - 1].compareTo(a[j]) > 0) {
    swap(a, j - 1, j);
    j--;
  }
}
/** Swaps entries `a[i]` and `a[j]`. Requires that `0 <= i < a.length` and `0 <= j <
  a.length`.*/
static <T> void swap(T[] a, int i, int j) {
    T temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

Back to…
Binary Search Trees

BST structure: Subtrees for everybody!



- Dotted circle is a subtree with null data and two null subtrees
- Enables us to represent empty trees
- Simplifies definition of BST methods

Code demo: BST

- fields: root (data), left, right
- constructor
- left() and right()

BST: find

```
private BST<T> find(T elem) { ... }
```

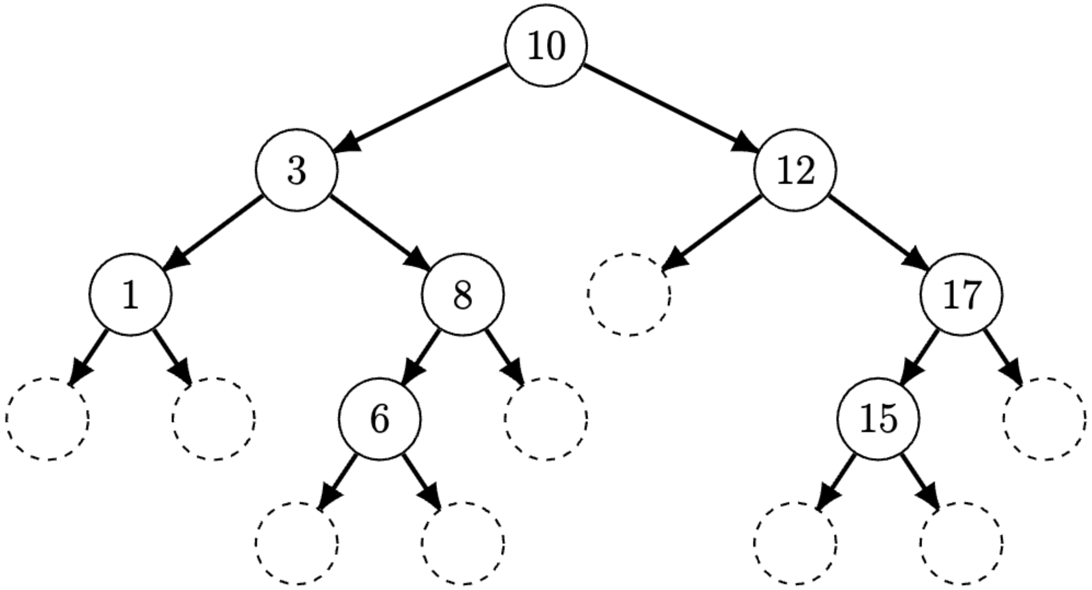
Recursion review – What does every recursive method have?

- Base Case
- Recursive Case

BST: find

Every recursive method has:

- Base Case(s)
  - Solve the smallest version of the problem
- Recursive Case(s)
  - solve a little increment of the problem
  - recur (call itself) on a smaller version of the problem



What do you think will be the base case(s) and recursive case(s) for find()?

```java
private BST<T> find(T elem) { ... }
```

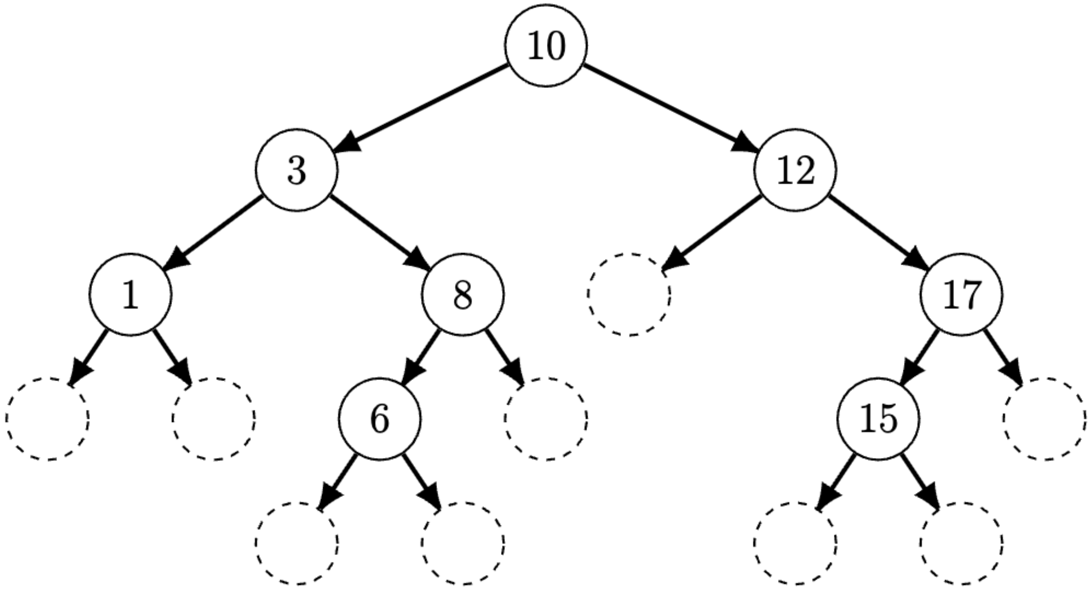
Code demo: BST.find()

Base case: root

Recursive cases

- Recur on left subtree

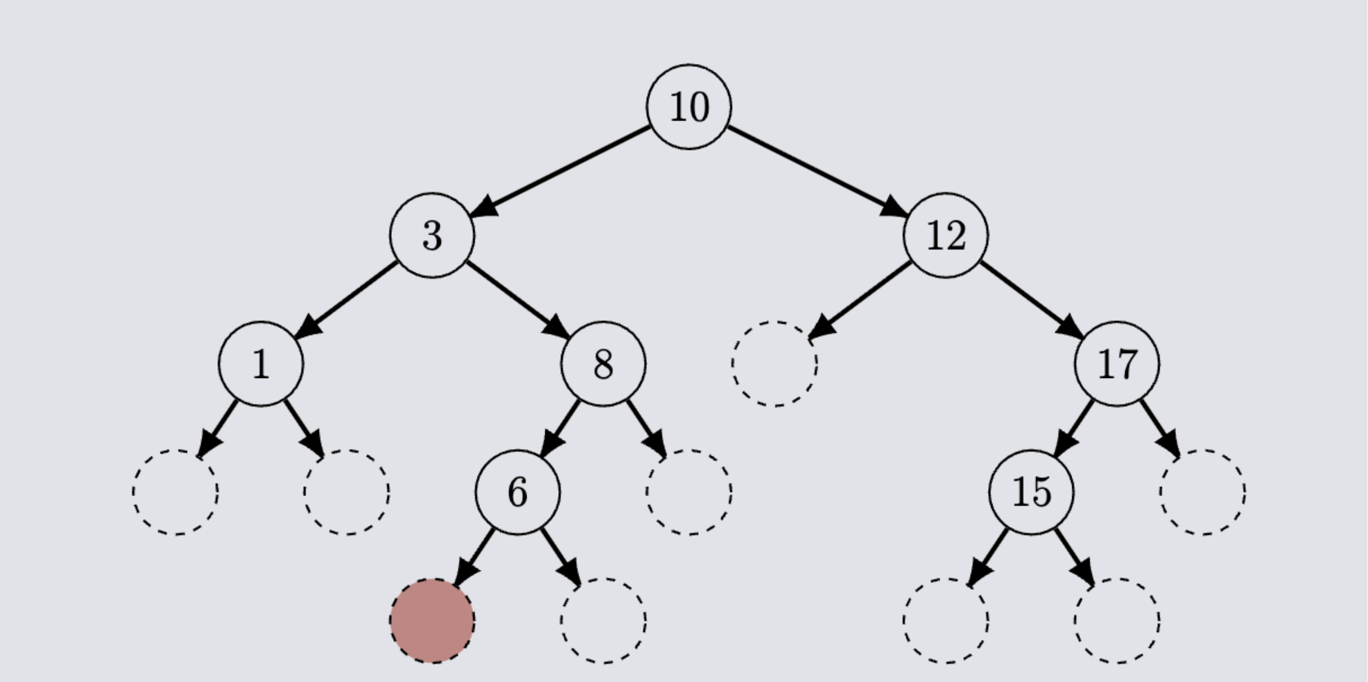
- Recur on right subtree



BST.add() -- intuition

```
private void add(T elem) { ... }
```

Use find() to figure out where it goes!

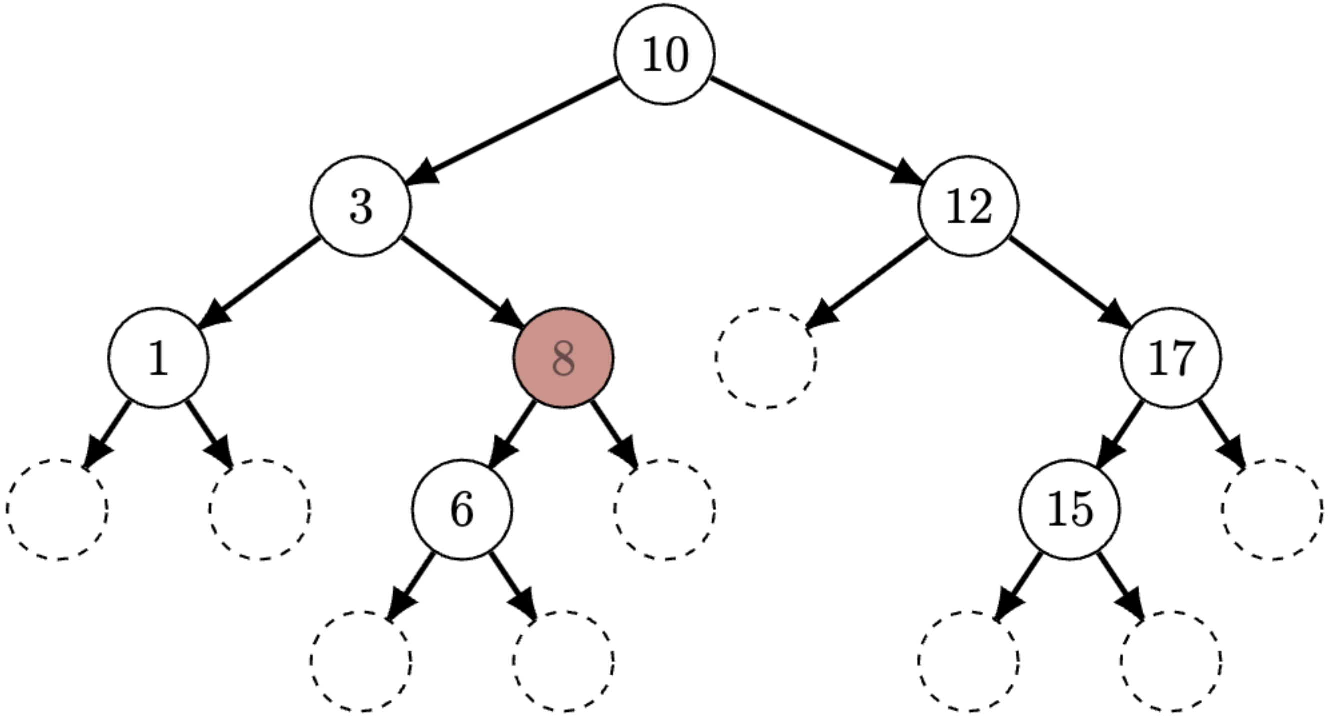


BST.remove() -- intuition

```
private void remove(T elem) { ... }
```

Use find() to get to the node to remove!

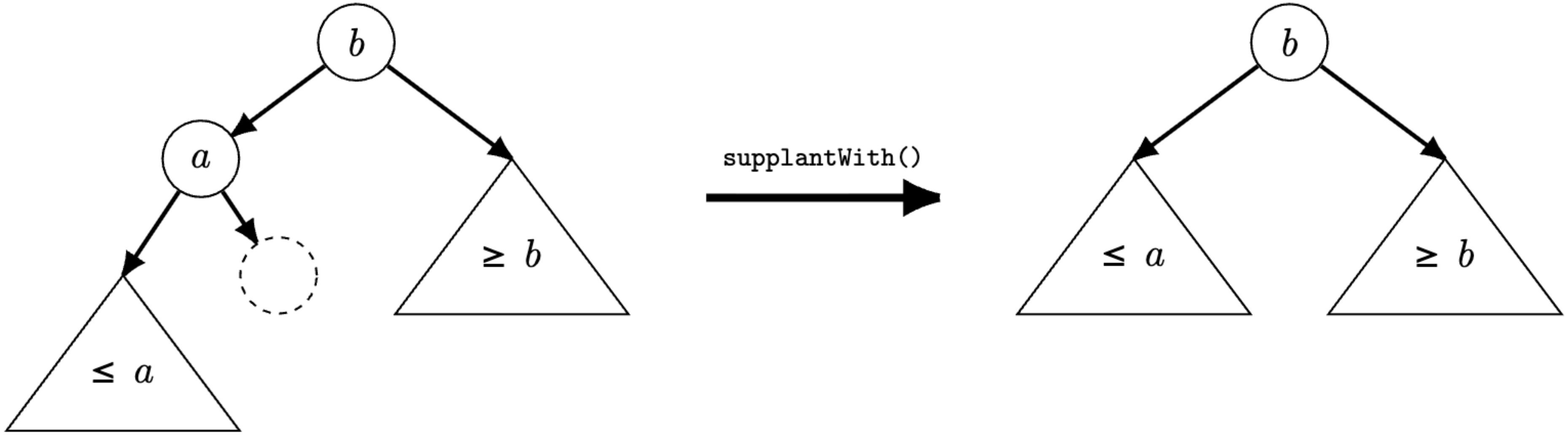
... then what?



BST.remove() -- intuition

```java
private void remove(T elem) { ... }
```

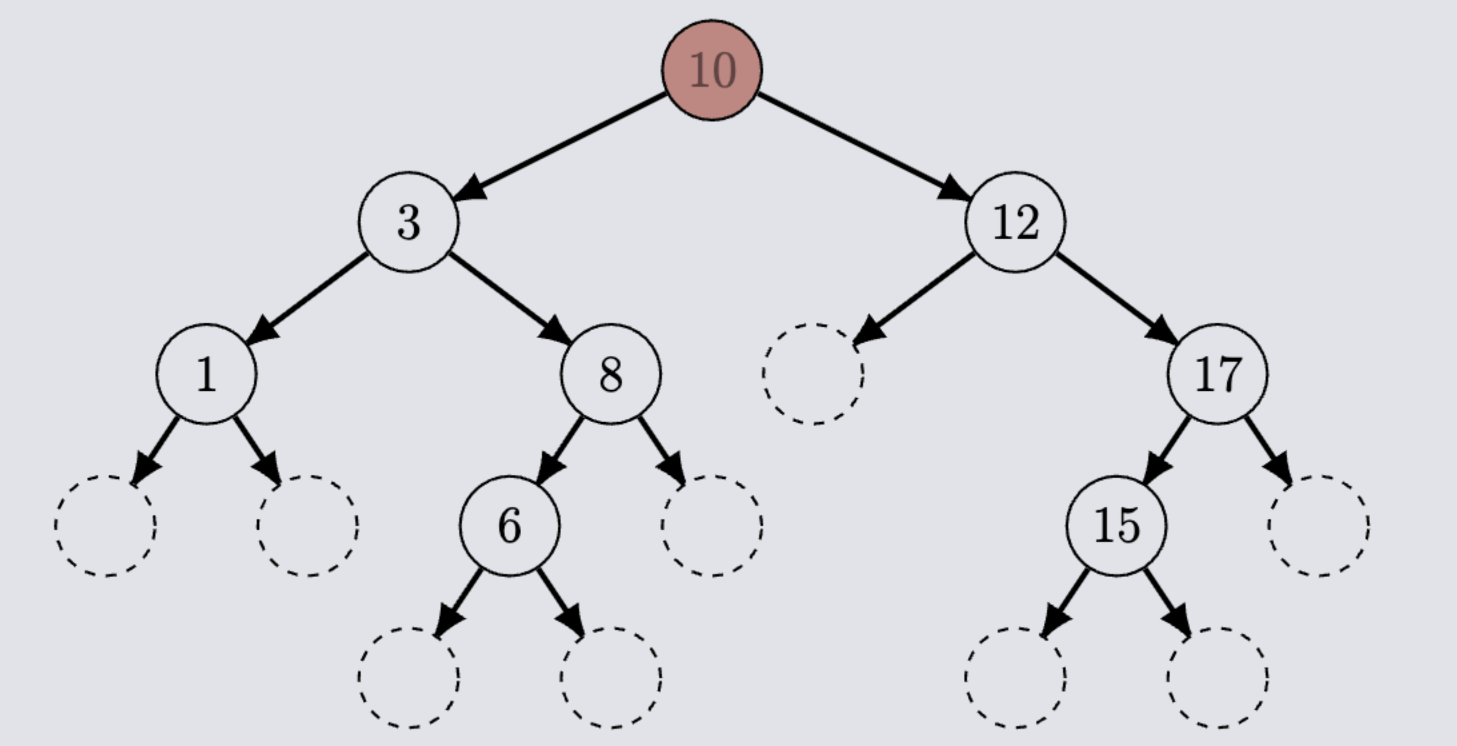
If right subtree is empty, we can supplant



BST.remove() -- intuition

```java
private void remove(T elem) { ... }
```

what if has two non-empty subtrees?



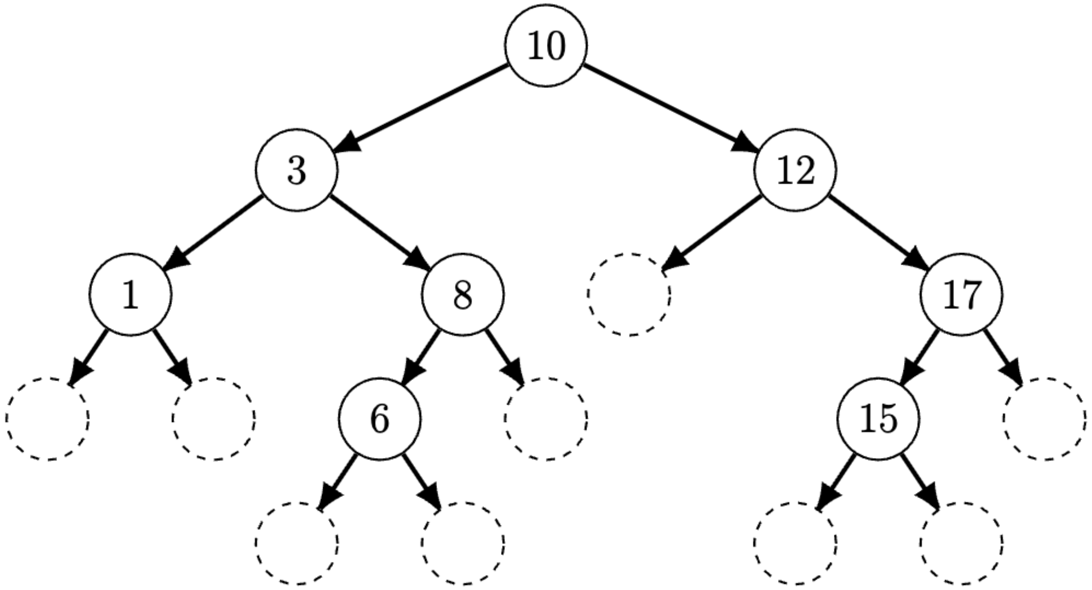
Binary Search Trees: Complexity

BST.find(): complexity

Base case: root

Recursive cases

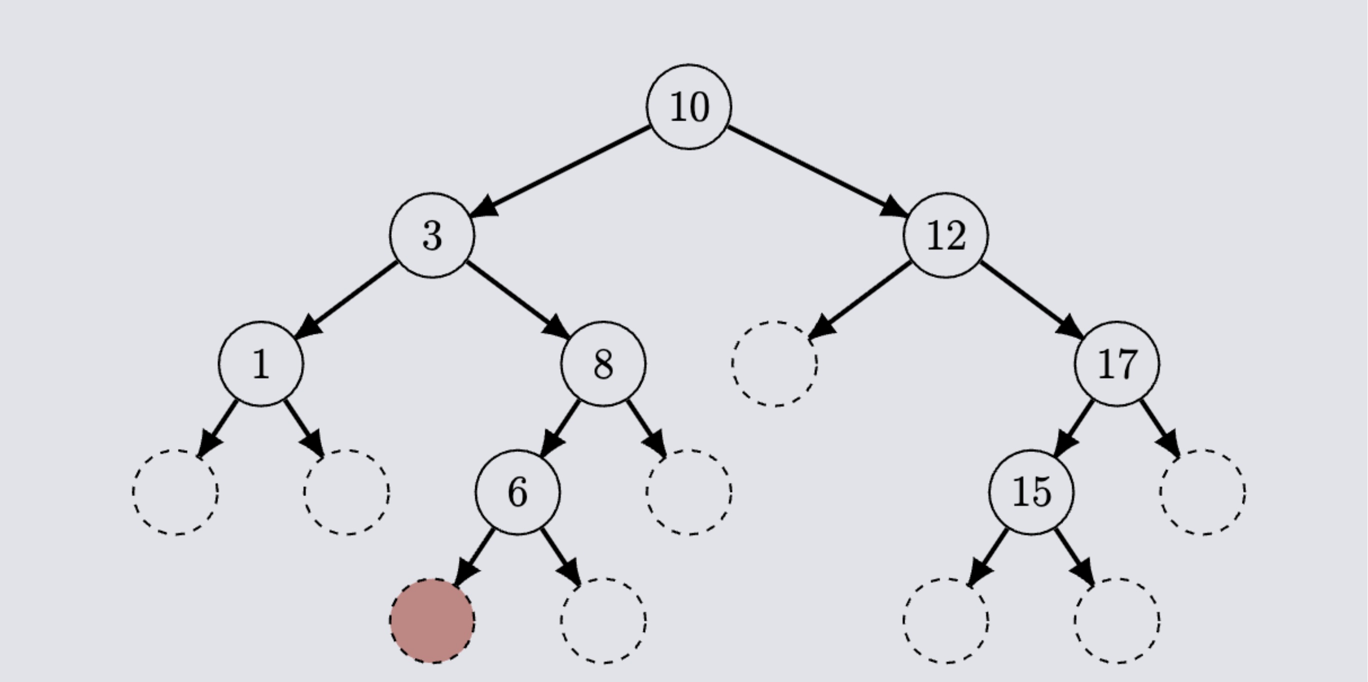
- Recur on left subtree
- Recur on right subtree



BST.add(): complexity

```java
private void add(T elem) { ... }
```

Use find() to figure out where it goes!

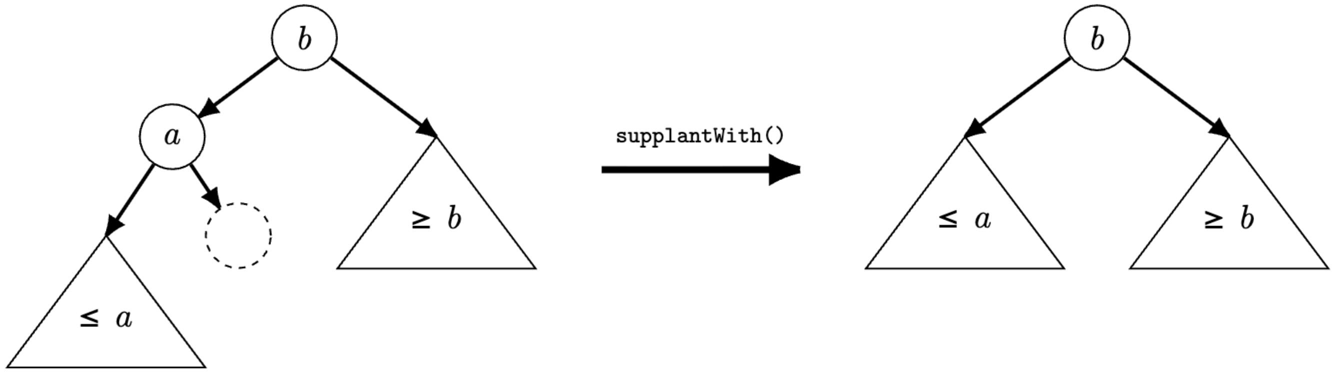


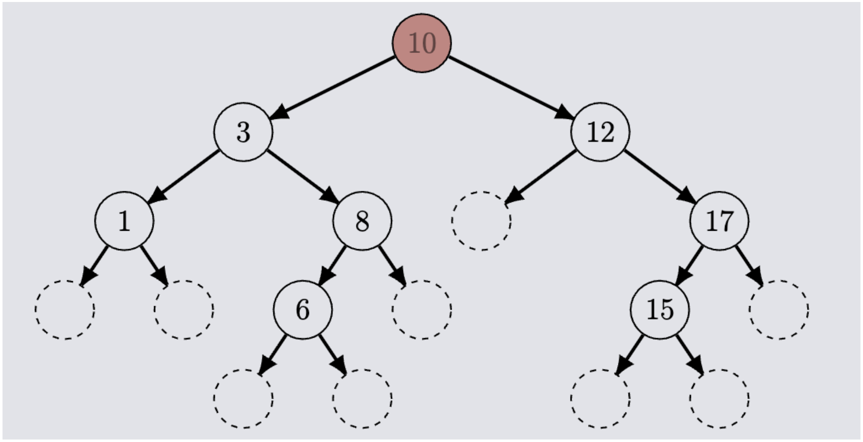
BST.remove() : complexity

```
private void remove(T elem) { ... }
```

If right subtree is empty, we can supplant

Otherwise, copy elements up until we can supplant





Metacognition

- Take 1 minute to write down a brief summary of what you have learned today

closing announcements to follow…

Announcements

- TA evaluations due Friday at 5pm
  - https://apps.engineering.cornell.edu/TAEval/survey.cfm --->
  - if you're not sure your TA's name for the eval, check the staff roster for their photo
- A8 released
  - long, hard, lots of reading (both code and English)
  - start early! go to office hours!
- Know your support resources
  - course website → About → Tips for Success