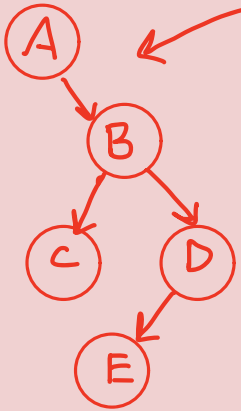
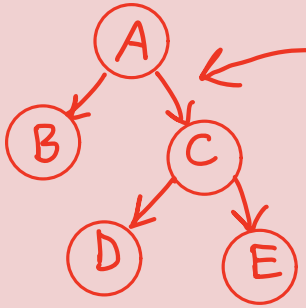
Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

A binary tree has pre-order traversal: **A, B, C, D, E**

Which of the following must be true about this tree?





A child of E would come after E.

B is in the left subtree. (A)

C is a descendant of **B**. (B)

E is a leaf. (C)

None of the above. (D)



Lecture 17: Binary Search Trees

CS 2110

October 22, 2026

Today's Learning Outcomes

62. Write and analyze recursive methods on binary trees.

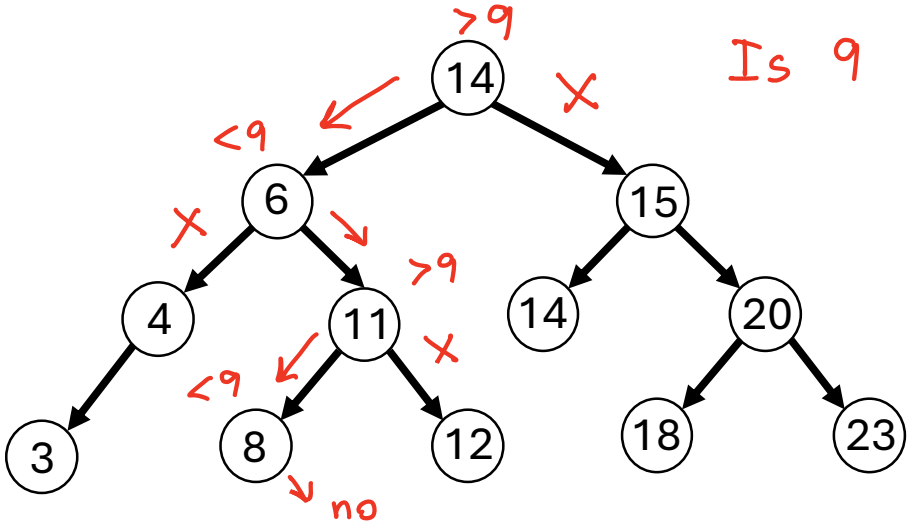
65. Describe the binary search tree invariant and determine whether it is satisfied by a given tree.

66. Explain the semantics of generic type bounds and write code that incorporates them.

67. Write modifying methods on a binary search tree that preserve its invariant.

68. Analyze the time/space complexities of a given method on a binary search tree.

A Nice Binary Tree



Is 9 in this tree?

In-Order Traversal: 3 4 6 8 11 12 14 14 15 18 20 23

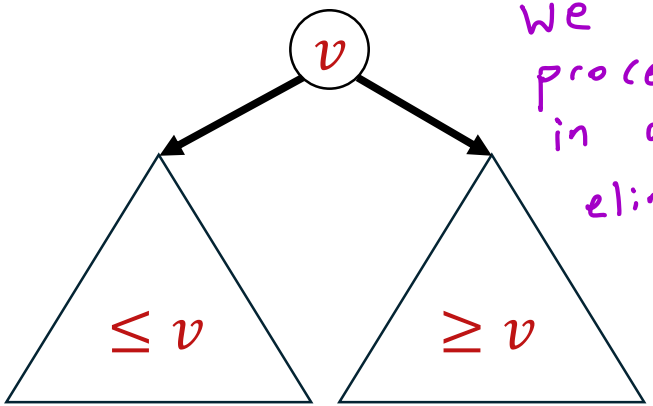
left subtree ≤ root

right subtree ≥ root

The Binary Search Tree Invariant

For every node v in a **Binary Search Tree** (BST),

- Each node ℓ in v's left subtree is $\leq v$
- Each node r in v's right subtree is $\geq v$



We can use a binary search-like procedure to quickly locate an element in a BST (one comparison per level eliminates entire subtree)

How do we compare elements that aren't numbers?

The Comparable<T> Interface

Types that implement the `Comparable<T>` interface can compare themselves to T elements.

(Typically, T is the same type.)

One Method: `int compareTo(T other)`

- returns negative number if "this < other"
- returns 0 if "this equivalent to other"
- returns positive number if "this > other"

To remember: Think "this − other"

** Read the documentation / lecture notes for more specific requirements.



Coding Demo: Making `Points` Comparable



The Comparator<T> Interface

`Comparable` objects know how to compare *themselves* to other objects.

`Comparators` are *separate* objects that are handed a pair of objects to compare.

"Give me two objects of type T, and I'll tell you which I think is greater."

Benefits of `Comparable`:

- simpler code
- type tells client a good way to compare it

Benefits of `Comparator`:

- more flexible
- multiple comparisons for same object

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

Suppose we initialize the following Comparator using a lambda expression:

* Warning: This Comparator is not consistent with equals()!

```
Comparator<Integer> cmp = (x,y) -> x % 5 - y % 5;
```

Which integer would cmp deem to be the "greatest" ?

greater = larger remainder mod 5

18	(A)	47	(C)
34	(B)	63	(D)

Generic Type Bounds

Our `BinaryTree<T>` class allowed us to store any type in its nodes.

A BST class must be able to compare the elements it scores so it can maintain its order invariant.

How do we enforce this? Generic type bounds

```
class BST<T extends Comparable<T>> extends BinaryTree<T>
```

T must be a subtype of (i.e. implement) Comparable<T>

CTRR: now we can call compareTo() on T objects



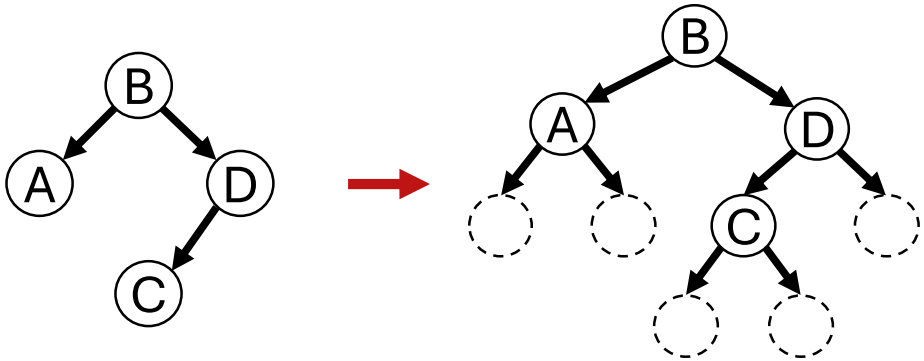
Coding Demo: Generic Sorting Methods



Coding Demo: The BST Class

Important Ideas:

- Class declaration with generic type bounds
- New "empty leaves" in state representation



The Most Important BST Helper Method

Exercise: Work with a partner to write this recursive method.

BST.java

```
/** Locates and returns a subtree whose root is `elem`, or the leaf child where `elem`
 *  would be located if `elem` is not in this BST. Requires that `elem != null`. */
private BST<T> find(T elem) {
    assert elem != null;
    if (root == null || elem.compareTo(root) == 0) {
        return this;
    } else if (elem.compareTo(root) < 0) { // elem is smaller, recurse on left
        return left.find(elem);
    } else { // elem is larger, recurse on right
        return right.find(elem);
    }
}
```

or elem.equals(root)

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333



What is the base case in your definition?

BST.java

```
/** Locates and returns a subtree whose root is `elem`, or the leaf child where `elem`
 *  would be located if `elem` is not in this BST. Requires that `elem != null`. */
private BST<T> find(T elem) {




}
```

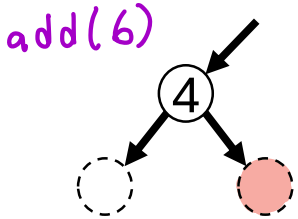
Adding an Element to a BST

Start by calling `find()`:

Adding an Element to a BST

Start by calling `find()`:

Case 1: If we end up at an "empty" leaf, put element there

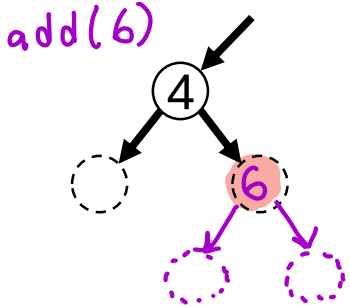


Adding an Element to a BST

Start by calling `find()`:

If we end up at a non-leaf, we're adding a duplicate element. Find a suitable "empty" leaf below for it.

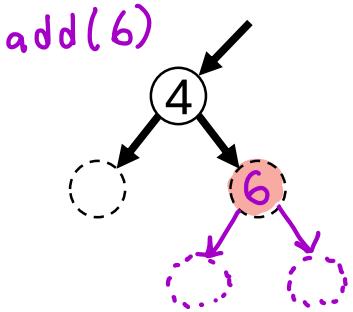
Case 1: If we end up at an "empty" leaf, put element there



Adding an Element to a BST

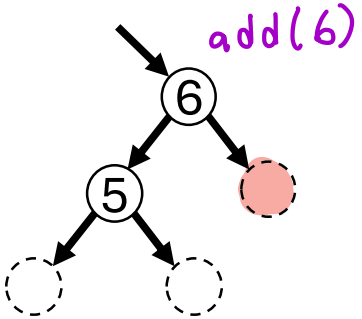
Start by calling `find()`:

Case 1: If we end up at an "empty" leaf, put element there



If we end up at a non-leaf, we're adding a duplicate element. Find a suitable "empty" leaf below for it.

Case 2: Empty Right Child

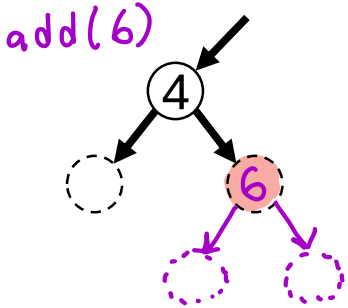


Adding an Element to a BST

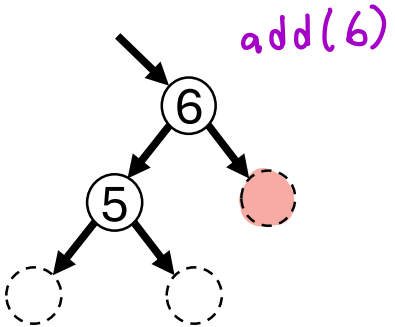
Start by calling `find()`:

If we end up at a non-leaf, we're adding a duplicate element. Find a suitable "empty" leaf below for it.

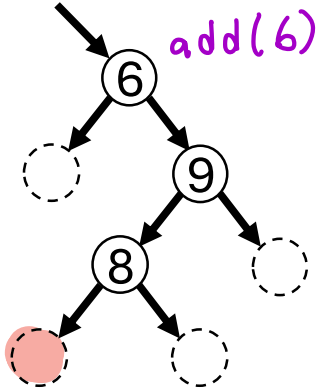
Case 1: If we end up at an "empty" leaf, put element there



Case 2: Empty Right Child



Case 3: Right Subtree
Locate its leftmost "empty" leaf





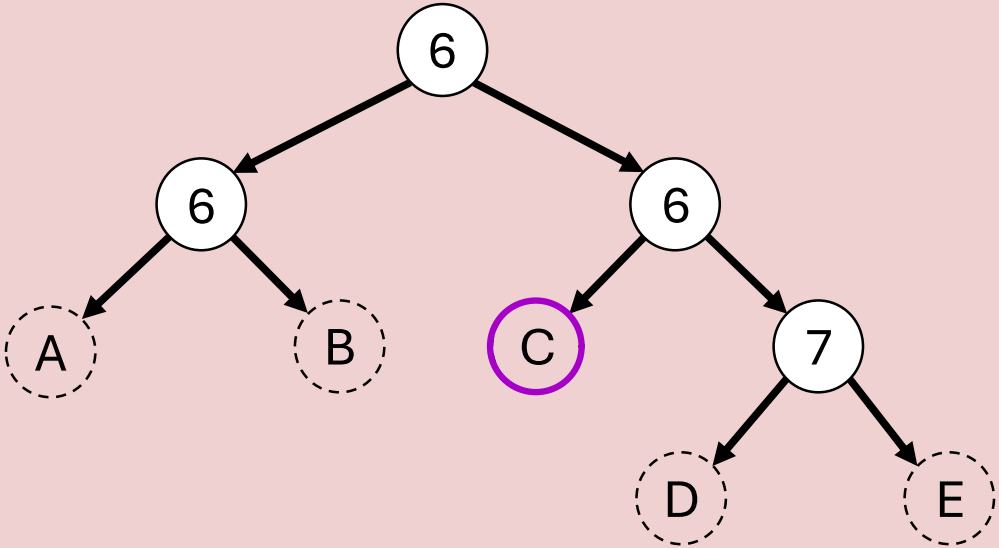
Coding Demo: `BST.add()`



Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

If we call add(6) on this BST with our add() definition, in which "empty" node will the new 6 be written?



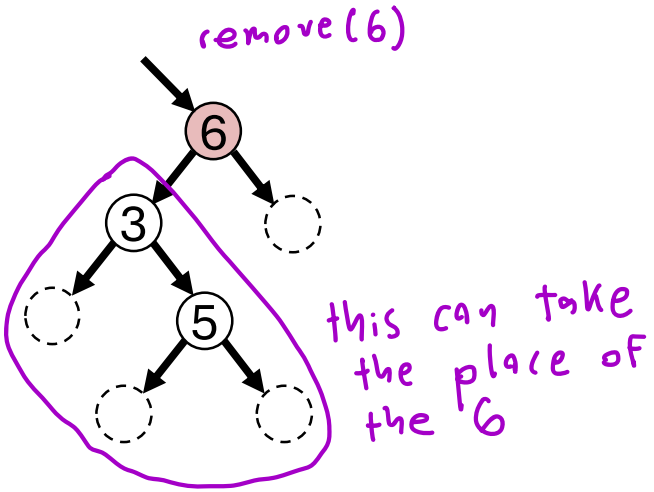
Removing an Element from a BST

Start by calling `find()`:

Removing an Element from a BST

Start by calling `find()`:

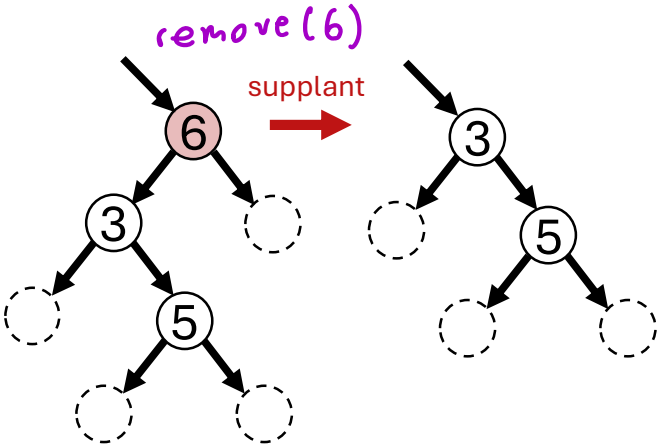
Case 1: Empty Right Child



Removing an Element from a BST

Start by calling `find()`:

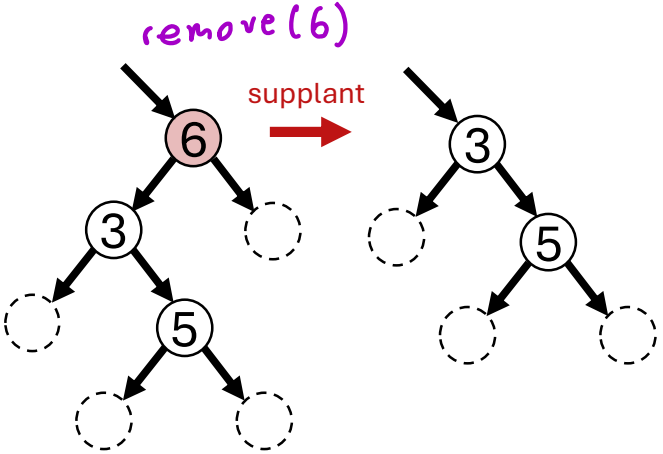
Case 1: Empty Right Child



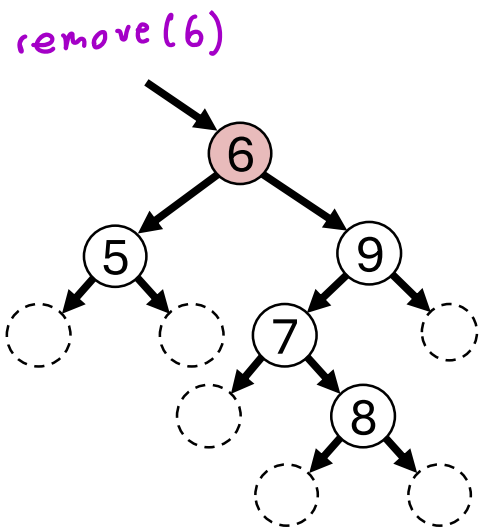
Removing an Element from a BST

Start by calling `find()`:

Case 1: Empty Right Child



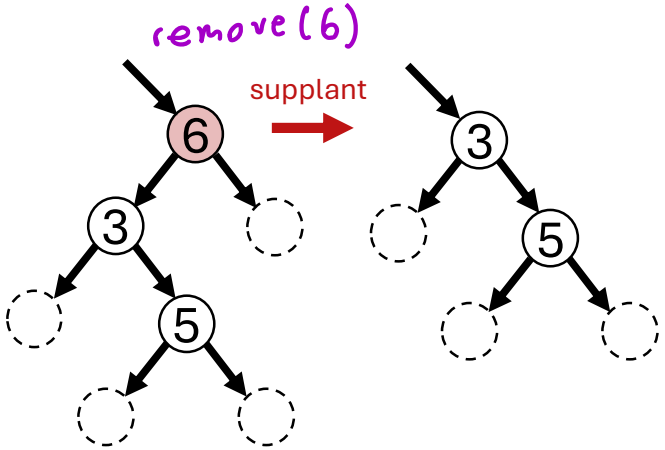
Case 2: Right Subtree



Removing an Element from a BST

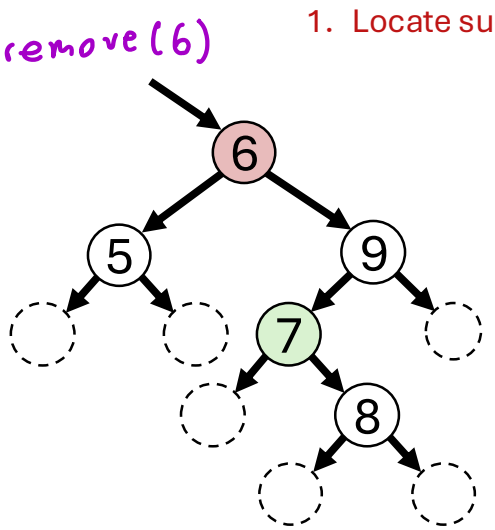
Start by calling `find()`:

Case 1: Empty Right Child



Case 2: Right Subtree

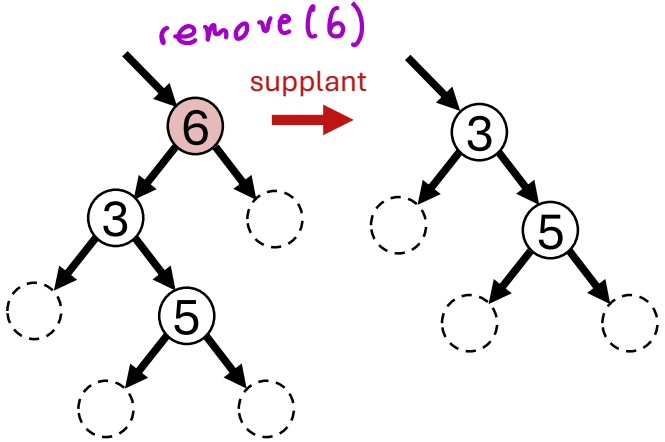
1. Locate successor in right subtree



Removing an Element from a BST

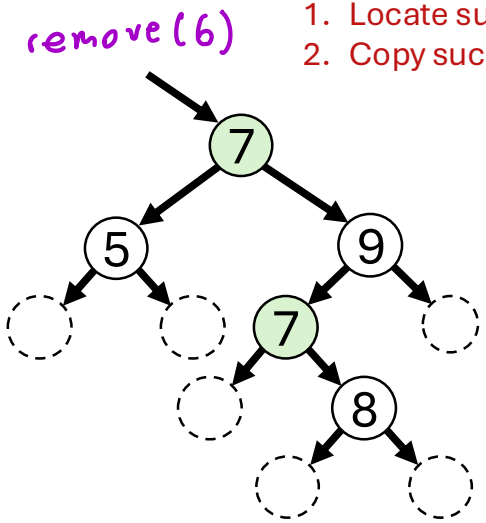
Start by calling `find()`:

Case 1: Empty Right Child



Case 2: Right Subtree

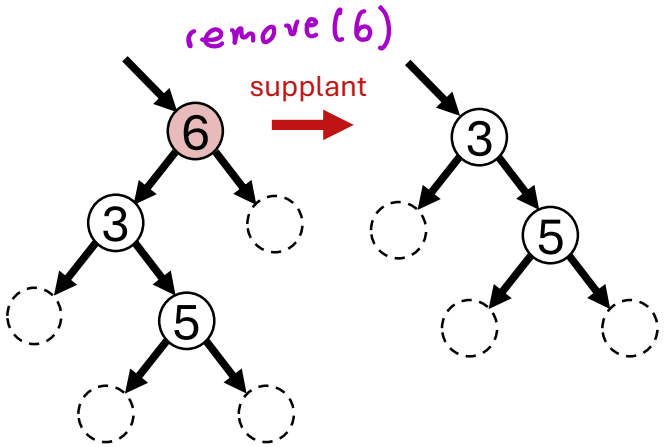
1. Locate successor in right subtree
2. Copy successor's value



Removing an Element from a BST

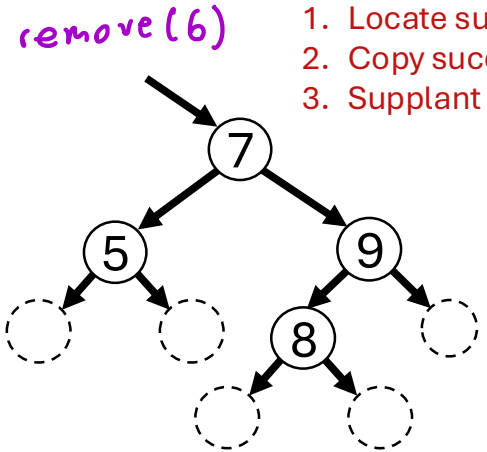
Start by calling `find()`:

Case 1: Empty Right Child



Case 2: Right Subtree

1. Locate successor in right subtree
2. Copy successor's value
3. Supplant successor's right subtree



BST Complexity Analysis

Every BST operation involves one find() call, up to one successorDescendant() call, and O(1) other work

Helper methods both do O(1) work per level so run in O(H) time
↖ height

⇒ All BST operations have worst case O(H) runtimes

Our recursive definitions require O(H) space for call frames, but iterative variants use O(1) space