


# Lecture 17: Binary Search Trees

CS 2110

October 22, 2026

# Today's Learning Outcomes

62. Write and analyze recursive methods on binary trees.

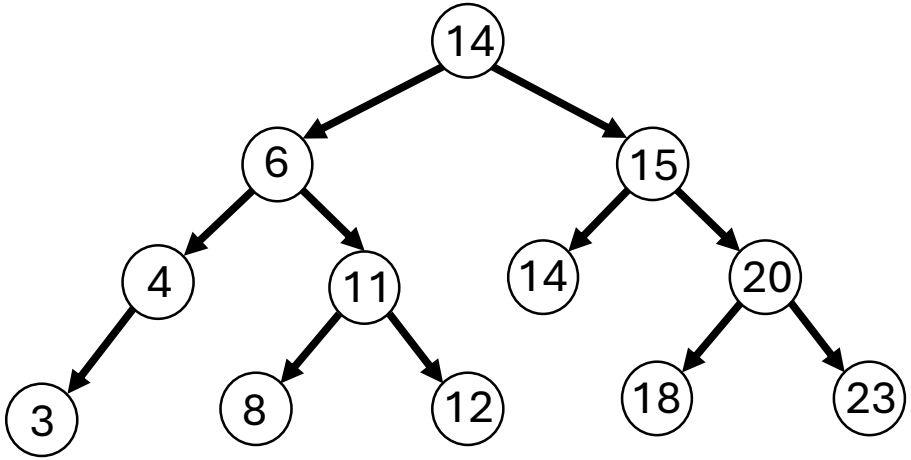
65. Describe the binary search tree invariant and determine whether it is satisfied by a given tree.

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

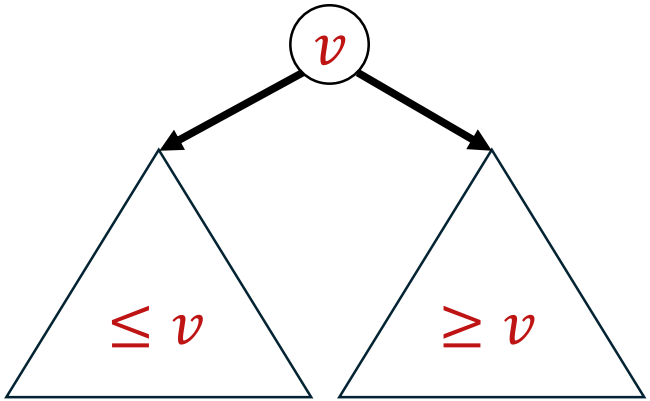



In-Order Traversal:

# The Binary Search Tree Invariant

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

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



# 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:**

** 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 others.

`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`:**

**Benefits of `Comparator`:**

# 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?*



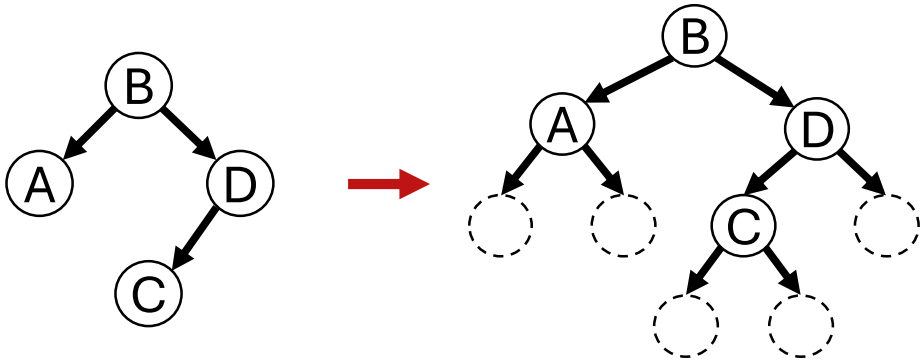
# Coding Demo: Generic Sorting Methods



# 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) {









}
```

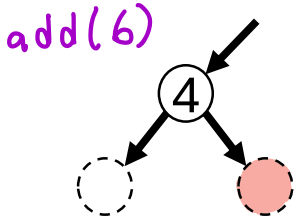
# Adding an Element to a BST

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

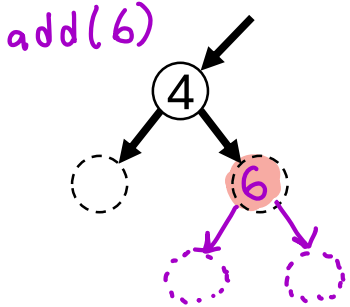


# Adding an Element to a BST

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.

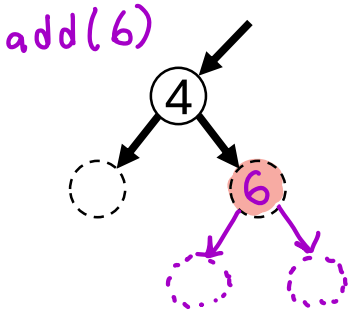
**Case 1:** If we end up at an "empty" leaf, put element there



# Adding an Element to a BST

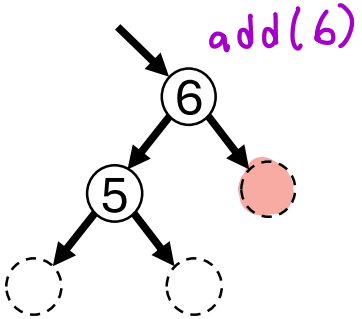
Start by calling `find()`:

**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

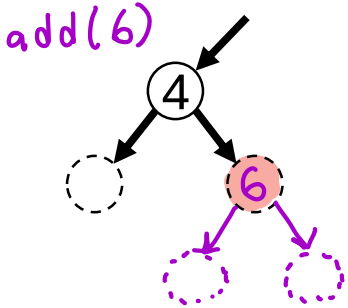


# Adding an Element to a BST

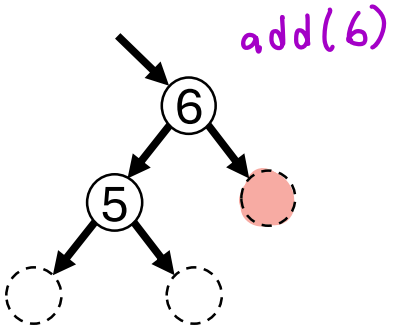
Start by calling `find()`:

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

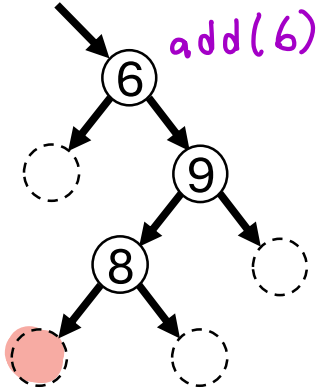
**Case 1:** If we end up at an "empty" leaf, put element there




**Case 2:** Empty Right Child




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





# Coding Demo: `BST.add()`



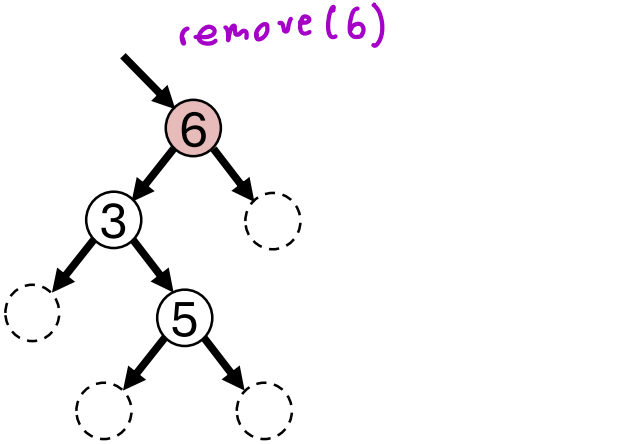
# Removing an Element from a BST

Start by calling `find()`:

# Removing an Element from a BST

Start by calling `find()`:

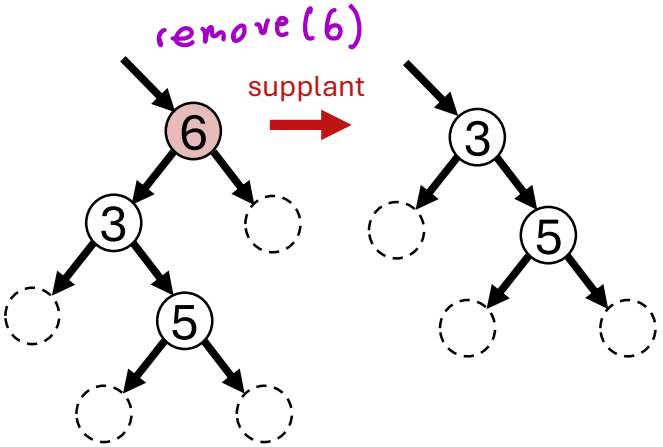
**Case 1:** Empty Right Child



# Removing an Element from a BST

Start by calling `find()`:

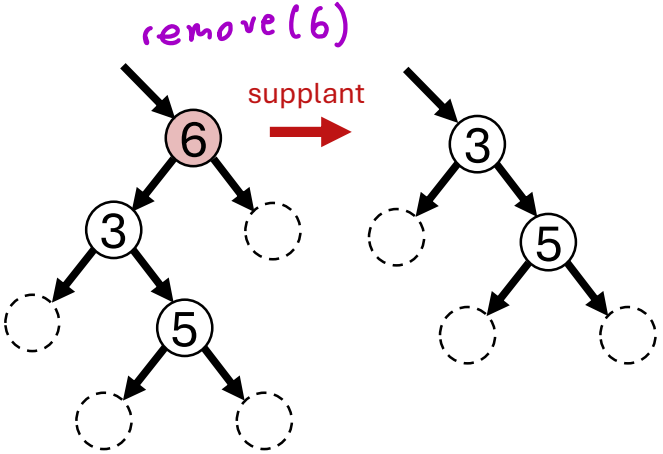
**Case 1:** Empty Right Child



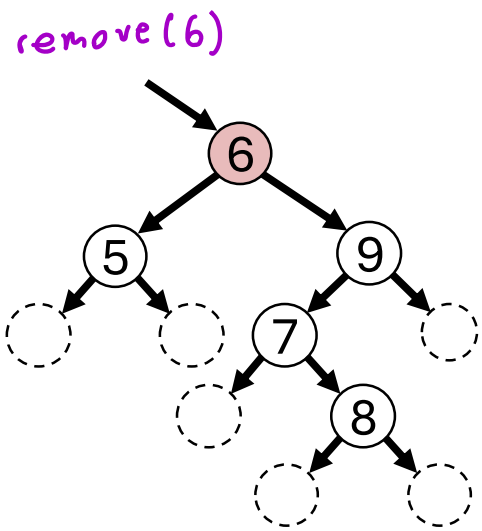
# Removing an Element from a BST

Start by calling `find()`:

**Case 1:** Empty Right Child



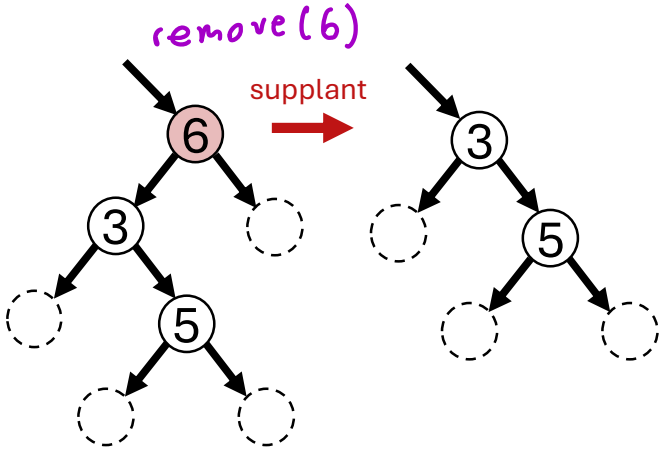

**Case 2:** Right Subtree



# Removing an Element from a BST

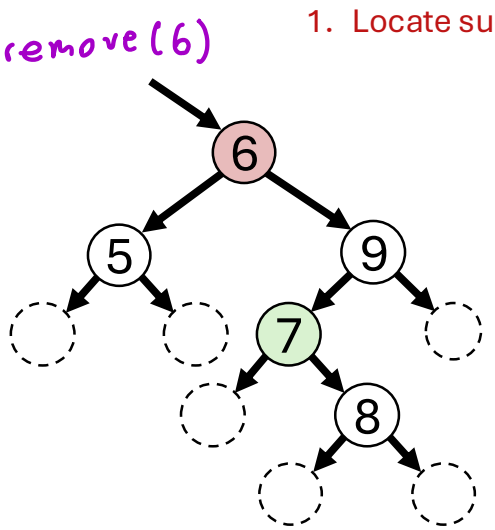
Start by calling `find()`:

**Case 1:** Empty Right Child




**Case 2:** Right Subtree

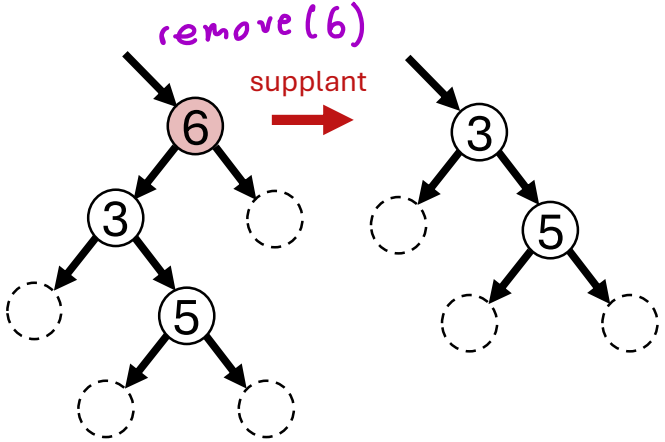
1. Locate successor in right subtree



# Removing an Element from a BST

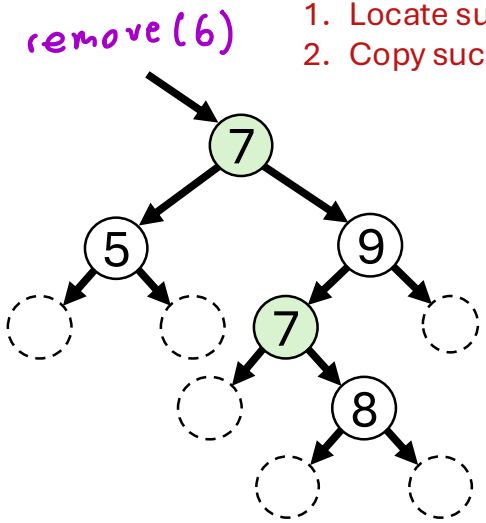
Start by calling `find()`:

**Case 1:** Empty Right Child




**Case 2:** Right Subtree

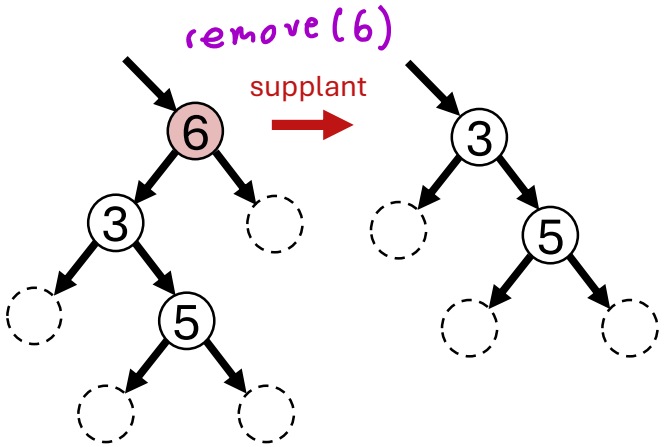
1. Locate successor in right subtree
2. Copy successor's value



# Removing an Element from a BST

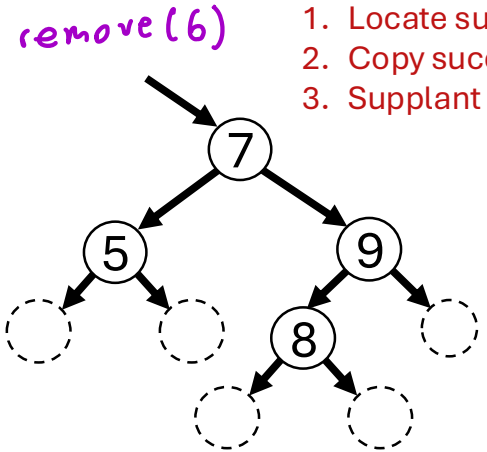
Start by calling `find()`:

**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