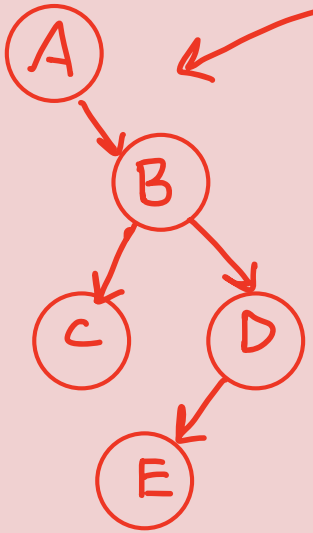
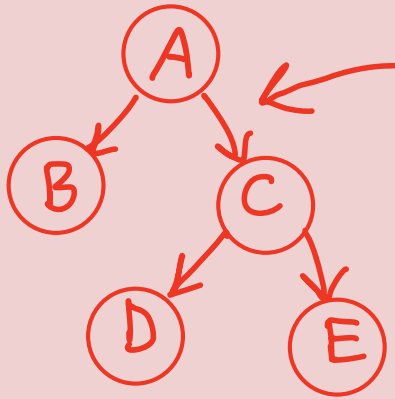
Poll Everywhere

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

March 19, 2026

Today's Learning Outcomes

67. Write recursive methods on binary trees.

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

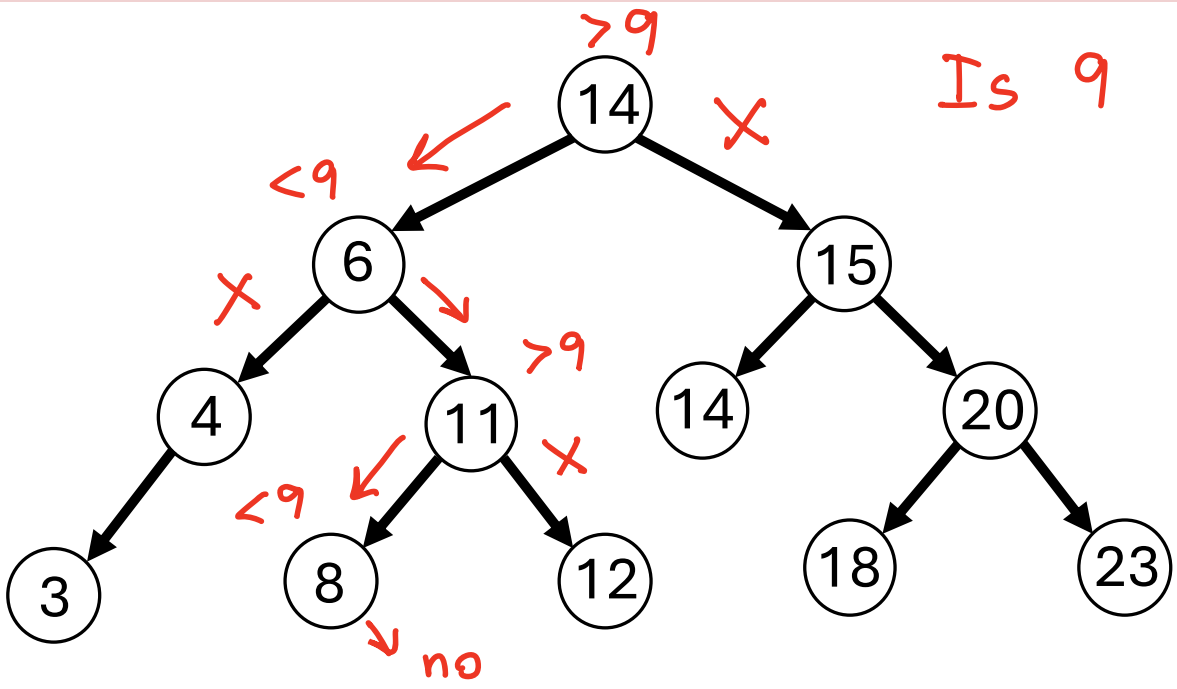
71. Describe the use cases for the `Comparable` and `Comparator` interfaces

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

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

74. 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

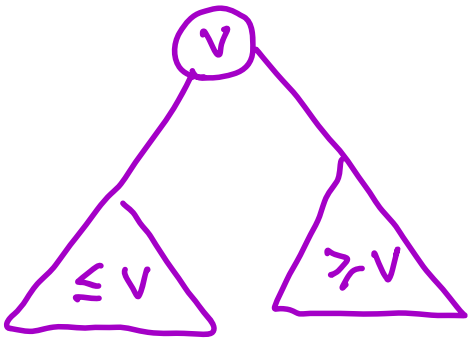
left subtree ≤ root

right subtree ≥ root

The Binary Search Tree Invariant

In a Binary Search Tree (BST), given any node v,

- 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 Interface

Types that implement the Comparable<T> interface are able to compare themselves to T elements.

(Typically T is 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"

Documentation outlines required properties

Coding Demo: Making `Points` Comparable

The Comparator Interface

Comparable objects know (internally) how to compare themselves to others.

Comparators are external objects that can be used to compare other objects.

"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/2110fa25 text 2110fa25 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 needs to be able to compare the elements it stores so it can maintain its order invariant.

Generic type bounds let us impose this requirement:

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

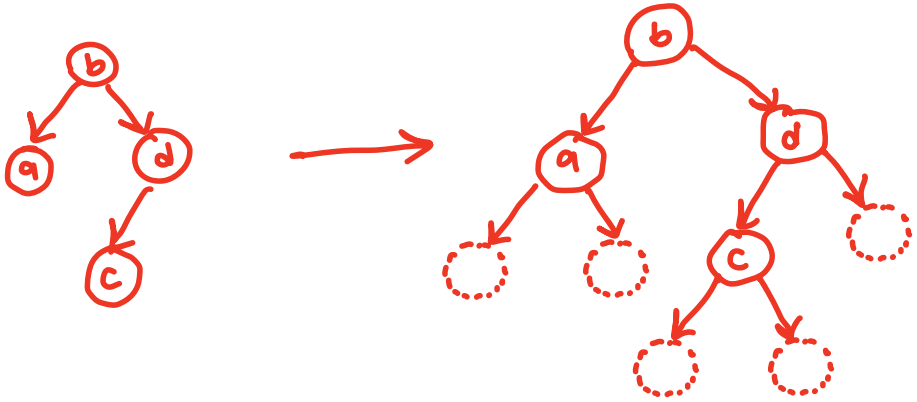
T must be a subtype of (i.e. implement) Comparable<T>

Coding Demo: Generic Sorting Methods

Coding Demo: The BST Class

Important Ideas:

- class declaration with generic type bound
- new "empty leaves" state representation



The Most Important BST Helper Method

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

```
/** 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);
    }
}
```

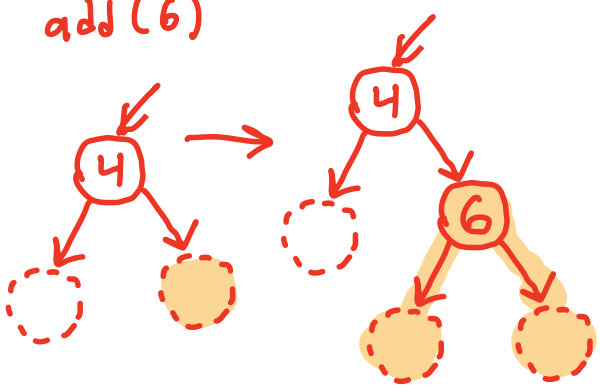
or elem.equals(root)

Adding an Element to a BST

Call find()

- If we end up at empty leaf, put element there

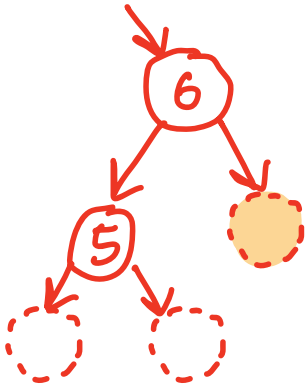
add(6)



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

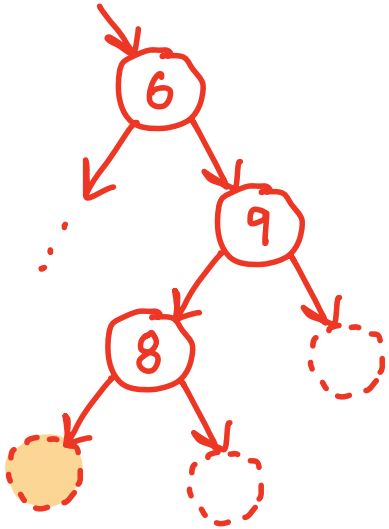
Case 1:
Empty Right Child

add(6)



Case 2:
Right Subtree

add(6)

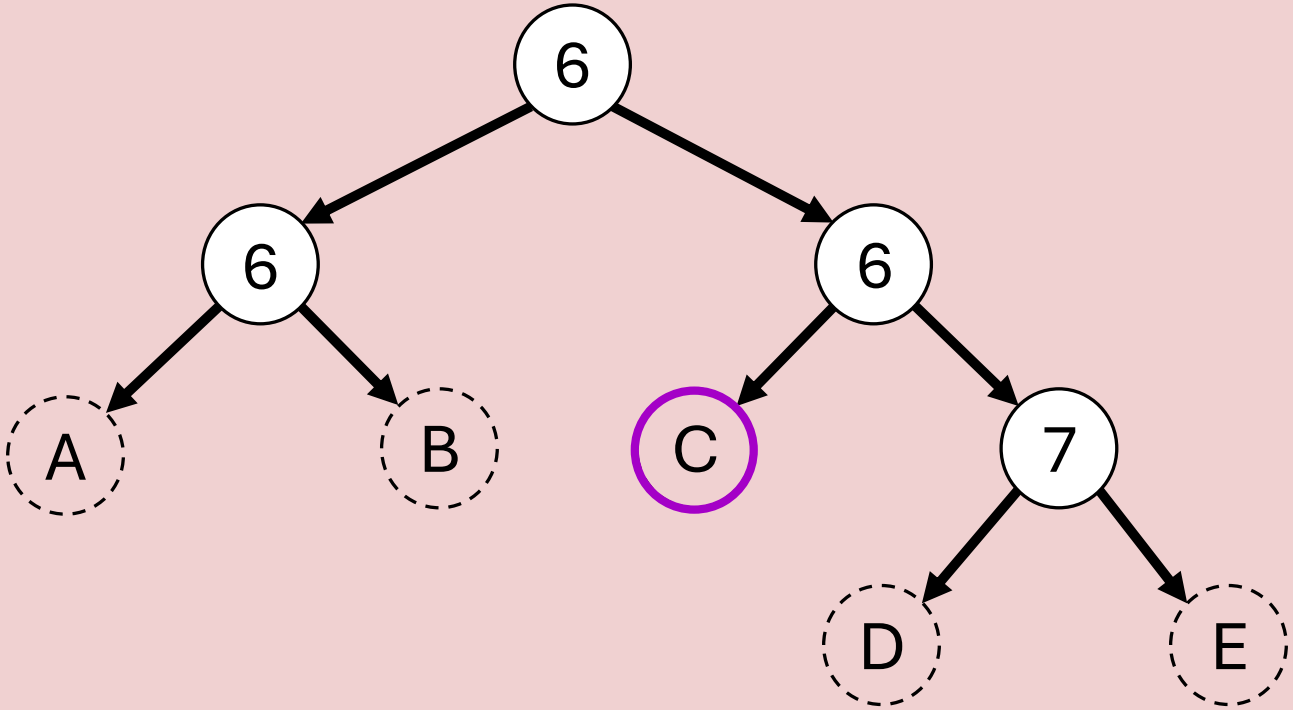


Coding Demo: `BST.add()`

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?

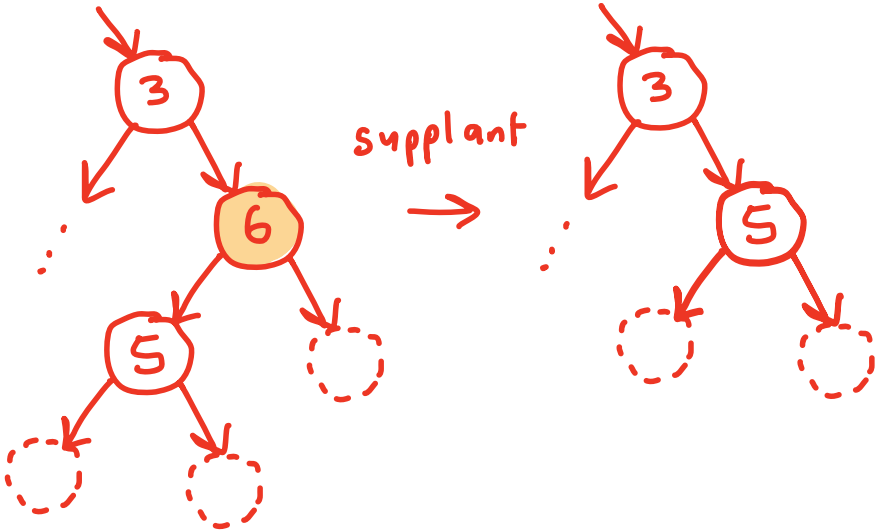


Removing an Element from a BST

Call find()

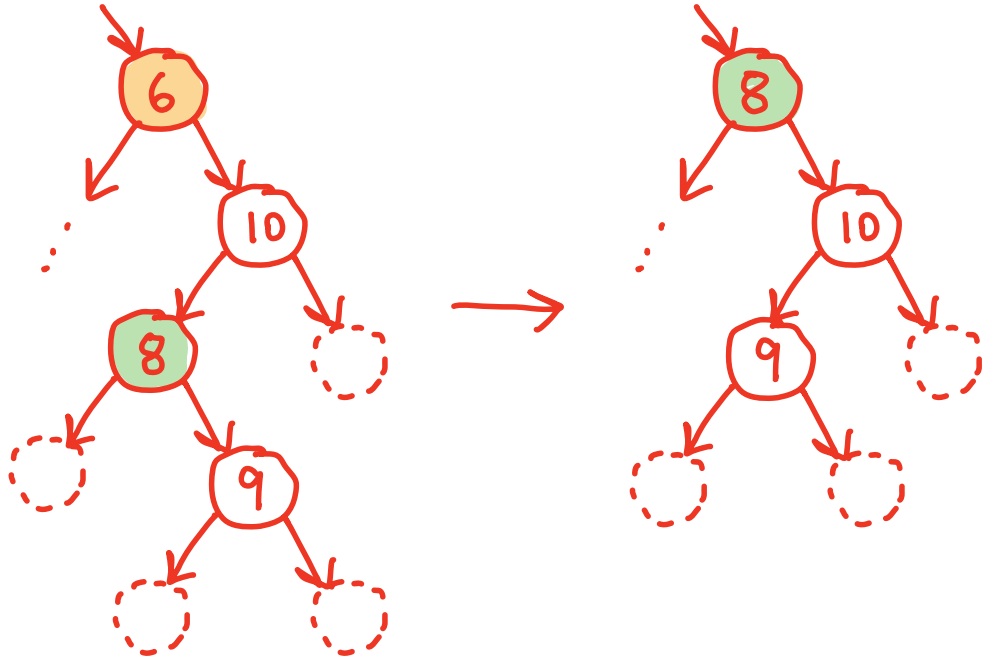
Case 1:

Empty Right Child



Case 2:

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

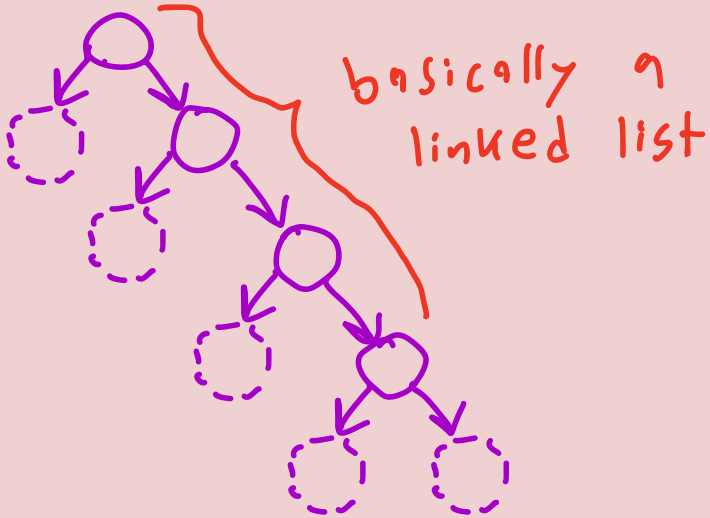
Our recursive definitions require O(H) space for call frames, but iterative variants use O(1) space

Poll Everywhere

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

All BST operations have a worst-case $O(H)$ runtime, where H is the tree's height. What's the tightest runtime bound in terms of N = the tree's size?

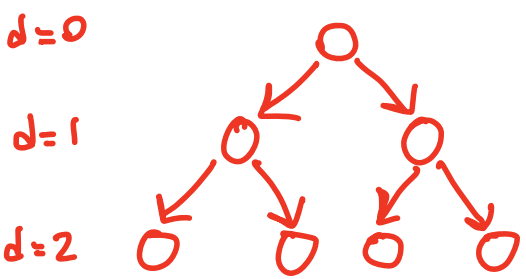
H can be as large as N



$O(\log N)$	**(A)**
$O(\sqrt{N})$	**(B)**
$O(N)$	**(C)**
$O(N \log N)$	**(D)**

Balanced BSTs

For a fixed height H, a binary tree can have at most



$$N \leq \sum_{d=0}^{H} 2^d = 2^{H+1} - 1 \quad \text{nodes.}$$

lower-bounded by

$$N = O(2^H) \Rightarrow H = \Omega(\log N)$$

Balanced binary trees have $H = O(\log N)$

Families of self-balancing BSTs maintain a balance invariant in $O(H) = O(\log N)$ time (take 3110 for more info)