Binary Search Trees

The Problem: Searchable Symbol Tables

Symbol Tables are lists of variables along with information about their scopes & types.

- The compiler generates and uses a symbol table to check the validity of the program and generate machine code.
  - Make sure no two variables are declared with the same name in the same scope
  - Make sure that every variable is declared before it is used

The compiler should be as fast as possible—you need to recompile your code every time you make a change!—so the symbol table should be quickly searchable.

Example: Searchable Symbol Tables

```
public class Counter {
        public int count;

        public Counter(int count) {
                this.count = count;
        }

        public void increment() {
                count++;
        }

        public void incrementBy(int amount) {
                count += amount;
        }

        ...
}
```

Symbol	Type	Scope
`Counter`	class	global
`count`	double	instance var
`increment`	void method	instance method
`incrementBy`	void method	instance method
`amount`	int	parameter

Understanding the Problem

- Representing the symbol table as a "table" is not well-defined—so let's define!
- We want to be able to:
  - Add a new symbol to the table (variable declaration)
  - Look up a symbol to see if it's already in the table (using a variable, variable declaration)
  - Remove a symbol from the table (leaving a specific scope in the program)
- ...and we want to be able to do all of the above quickly.

The Symbol Class

```
public record Symbol(String name, String type) implements Comparable<Symbol> {
        @Override
        public int compareTo(Symbol other) {
                return this.name.compareTo(other.name);
        }
}
```

Formalizing the Interface

Goal	Method	Notes
Add a new symbol to the table	`boolean add(Symbol s)`	Careful! What happens if we already have such a Symbol?
Look up a symbol	`boolean contains(Symbol s)`	
Remove a symbol	`boolean remove(Symbol s)`	How do we decide which Symbols to remove?

Do We Need a New Data Structure?

Is there any reason to build something new to solve this problem, or can we leverage something we already know how to use?

- Lists?
  - `contains` is $O(n)$ if the list is unsorted, which is bad since we'll have to look up symbols a lot!
  - `contains` is $O(\log n)$ if the list is sorted, but then `add` is even slower to maintain the sorted order

Do We Need a New Data Structure?

Is there any reason to build something new to solve this problem, or can we leverage something we already know how to use?

- Stacks? Queues?
  - These are actually vital for the most efficient real world compilers to help define scope in a clever way, but that's beyond us here.
  - In our case, not really clear how to support `contains` in a Stack/Queue.
- Trees?
  - General binary trees don't offer anything better than lists a priori
  - Huffman Trees—too specific!

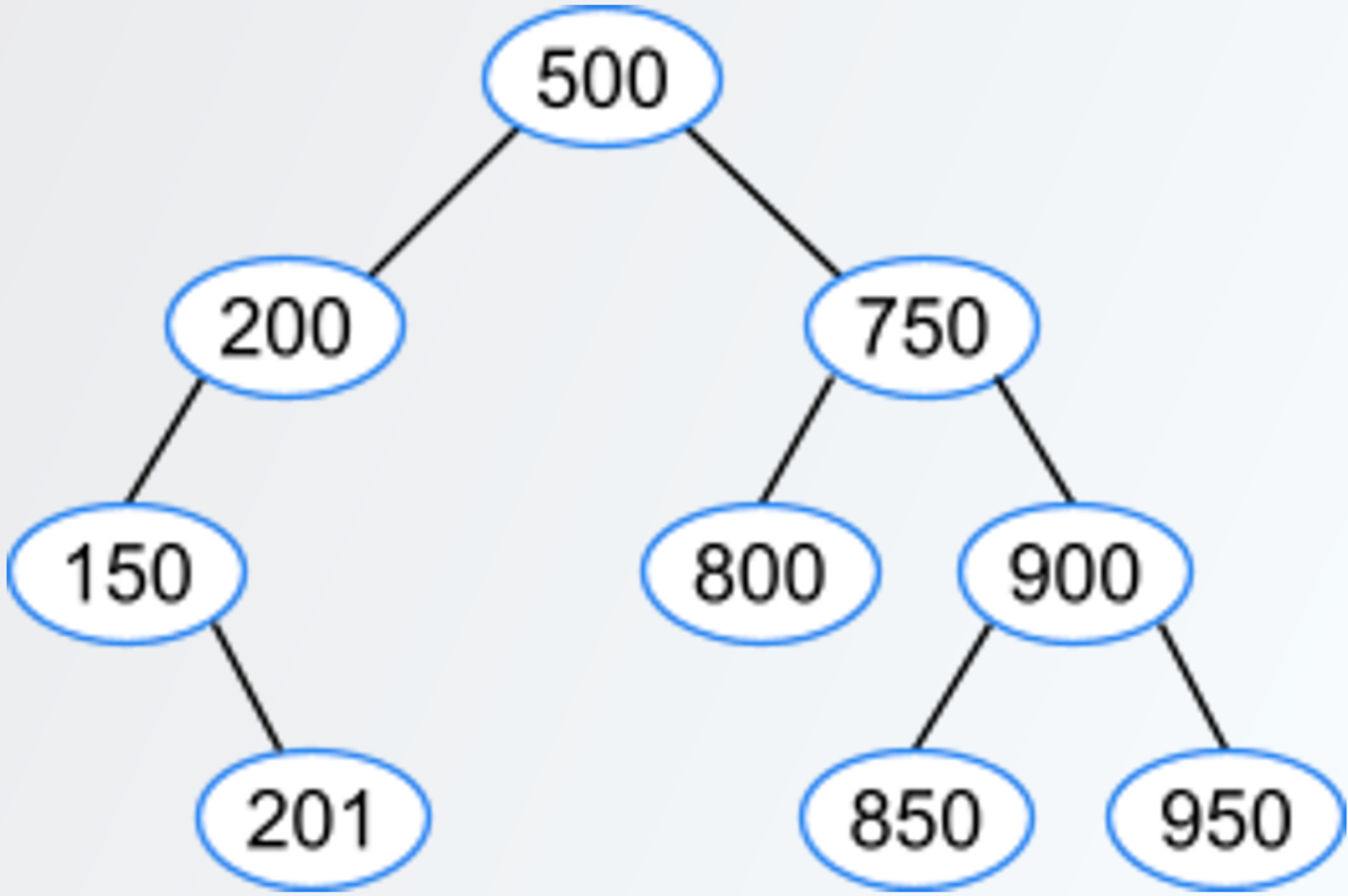
Our New Data Structure

Binary Search Trees are Binary Trees with the **Binary Search Tree property:**

- All nodes stored in the left subtree of a node whose key value is K have key values less than or equal to K
- All nodes stored in the right subtree of a node whose key value is K have key values greater than K
- ...and all subtrees of the root obey the BST property too, recursively

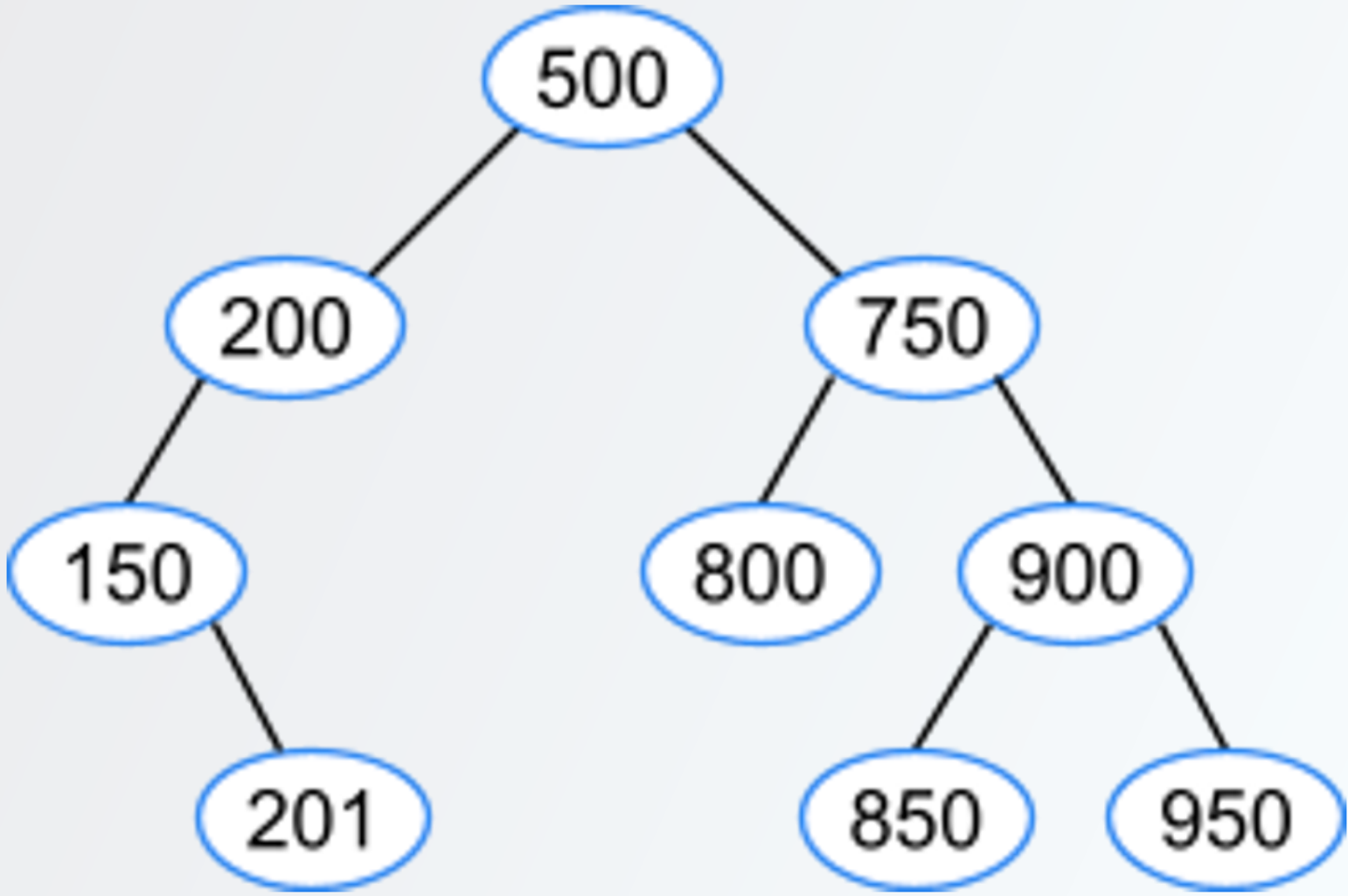
Which Subtrees of This Tree Are BSTs?

- The tree rooted at the Node of 900?
- The tree rooted at the Node of 750?
- The tree rooted at the Node of 500?
- The tree rooted at the Node of 201?



Which Subtrees of This Tree Are BSTs?

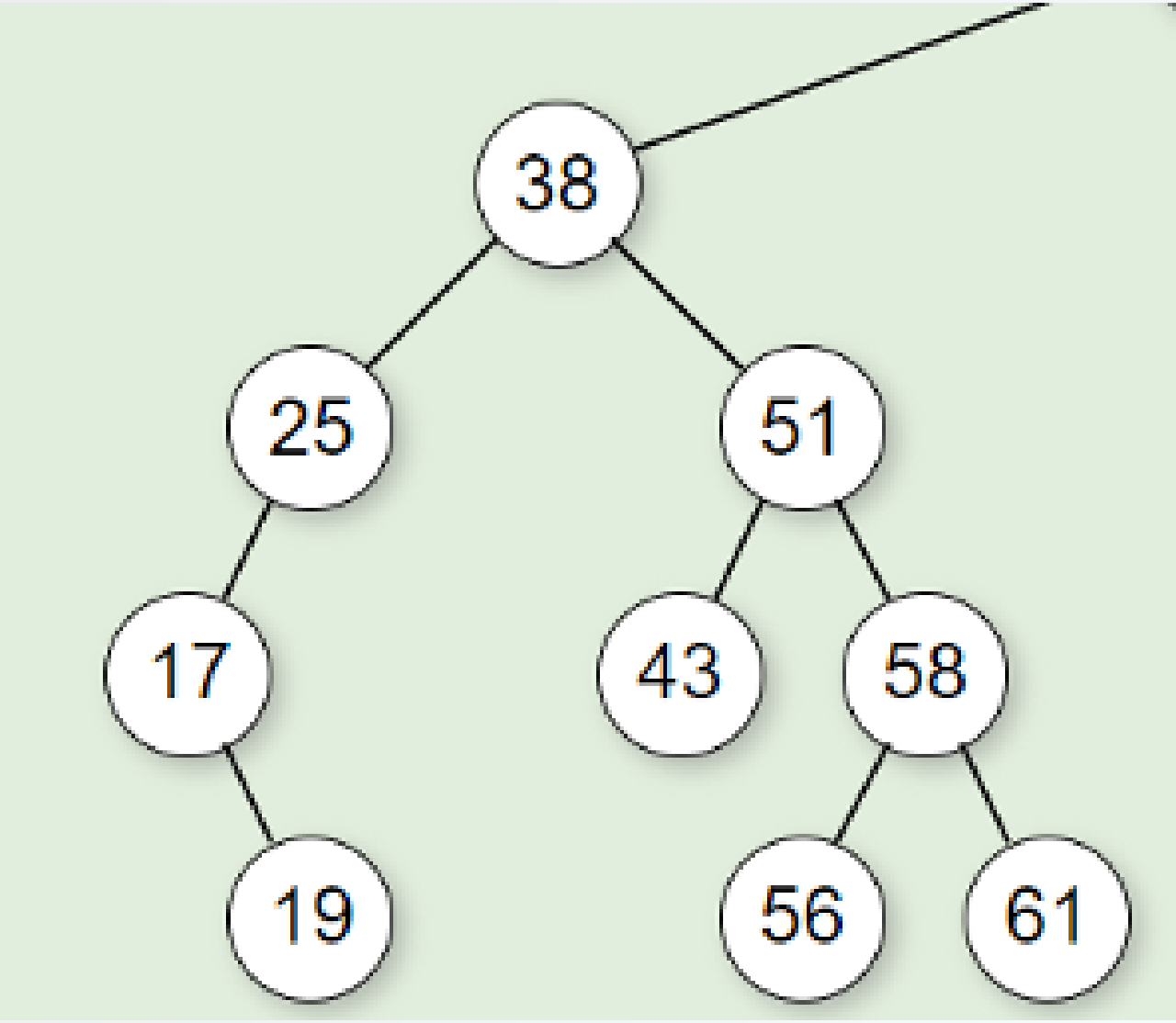
- The tree rooted at the Node of 900? ✅
- The tree rooted at the Node of 750? ❌
- The tree rooted at the Node of 500? ❌
- The tree rooted at the Node of 201? ✅



Inorder Traversal

Recall: Inorder Traversal visits the left subtree, then the current node, then the right subtree.

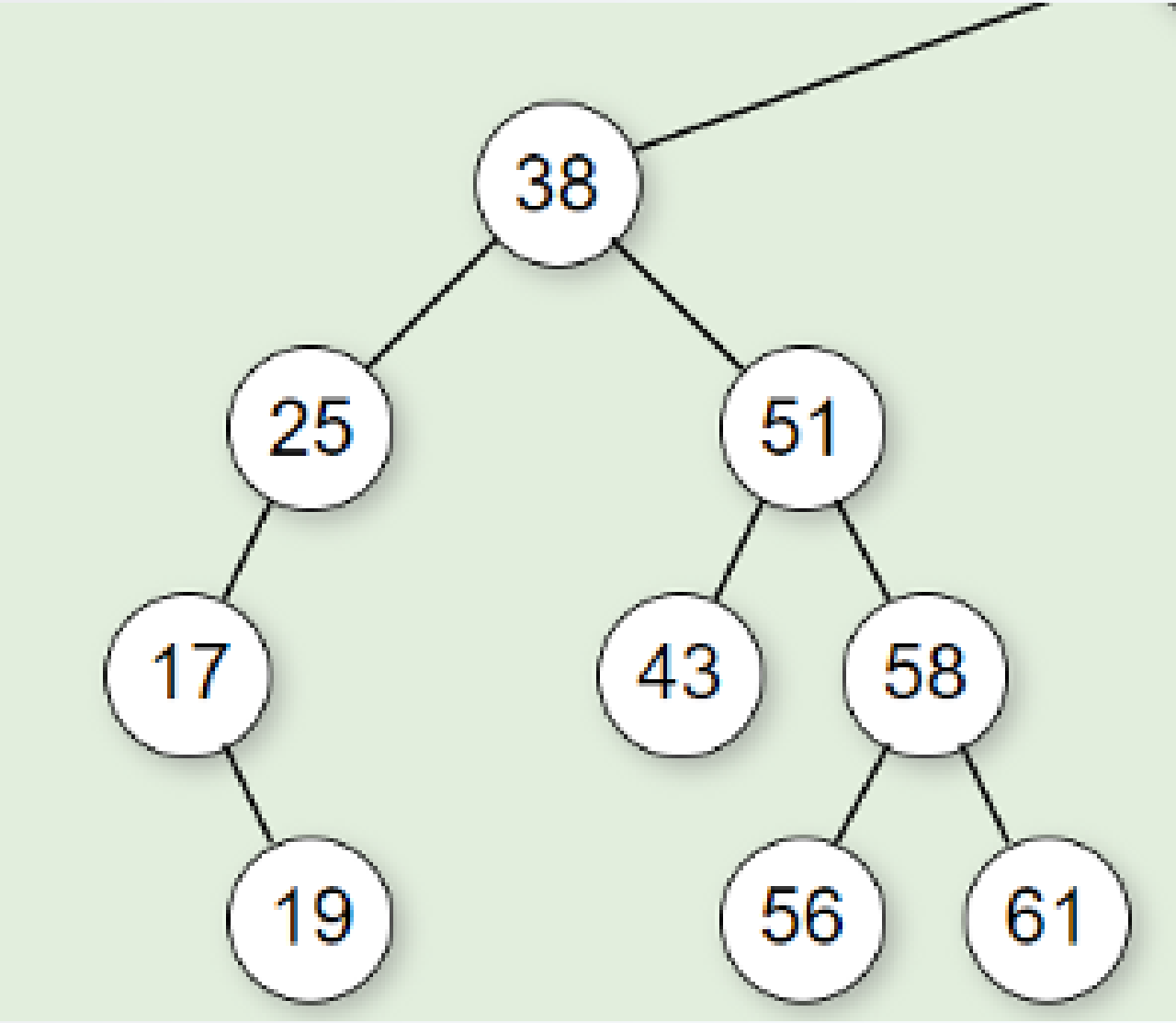
Returns the records (stored in the nodes) in sorted order from lowest to highest



Inorder Traversal

For the tree rooted at 38, the inorder traversal is:

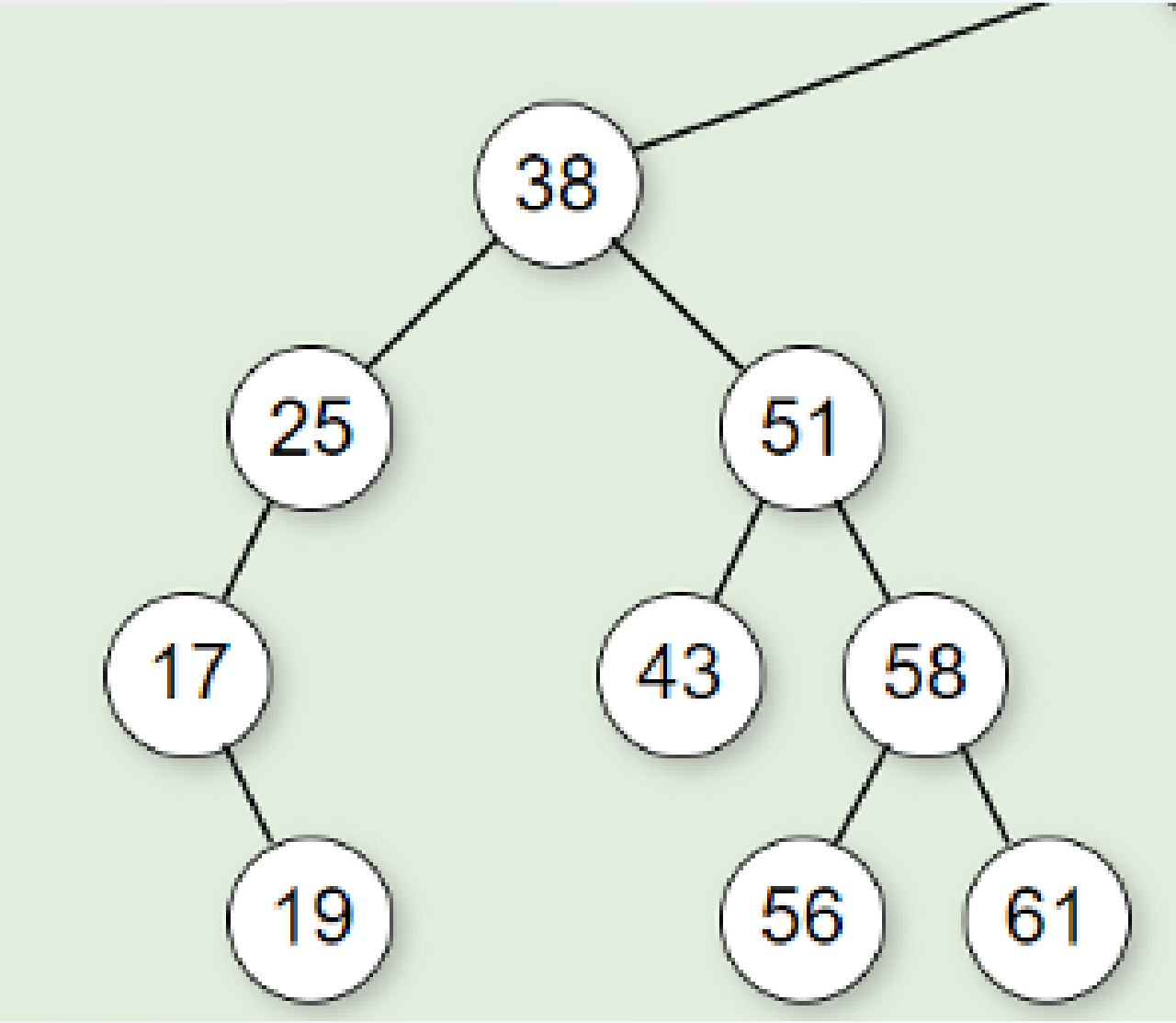
Visit the left subtree (rooted at 25), then the current node (38), then the right subtree (rooted at 51).



Inorder Traversal

For the tree rooted at 25, the inorder traversal is:

Visit the left subtree (rooted at 17), then the current node (25), then the right subtree (none!).

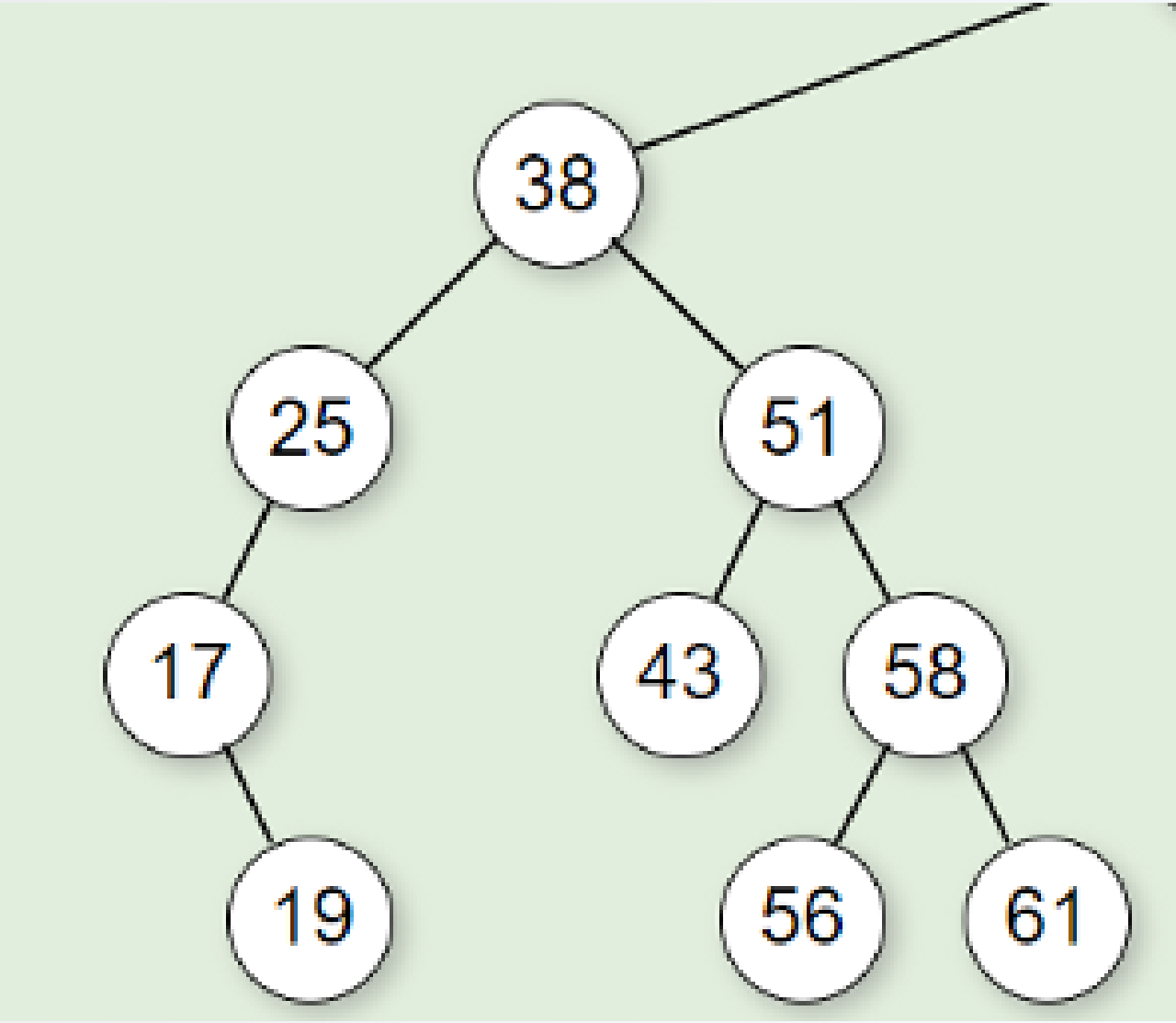


Inorder Traversal

For the tree rooted at 17, the inorder traversal is:

Visit the left subtree (none), then the current node (17), then the right subtree (19).

🖨️➡️ 17

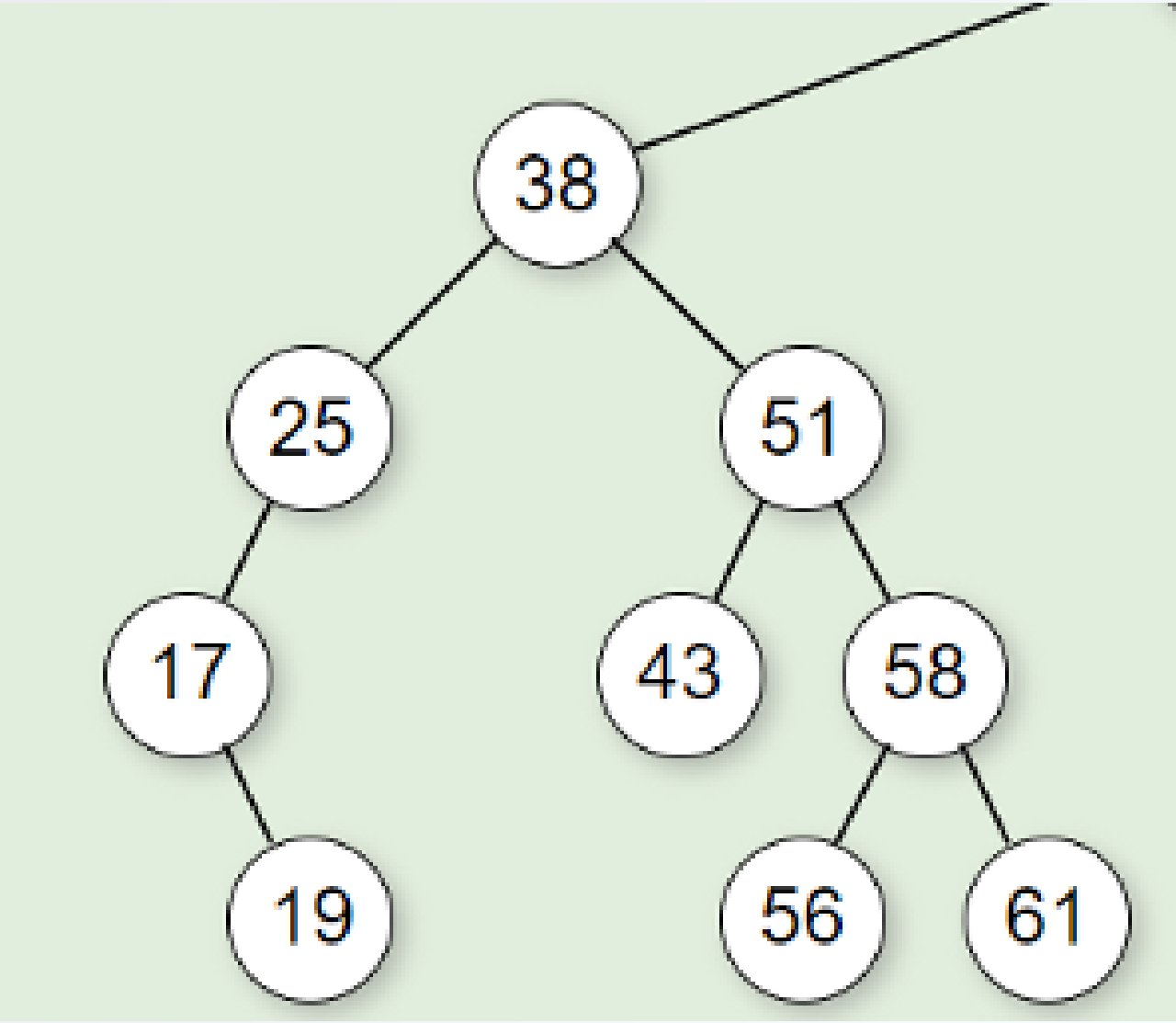


Inorder Traversal

For the tree rooted at 19, the inorder traversal is:

Visit the left subtree (none), then the current node (19), then the right subtree (none).

🖨️➡️ 17 19

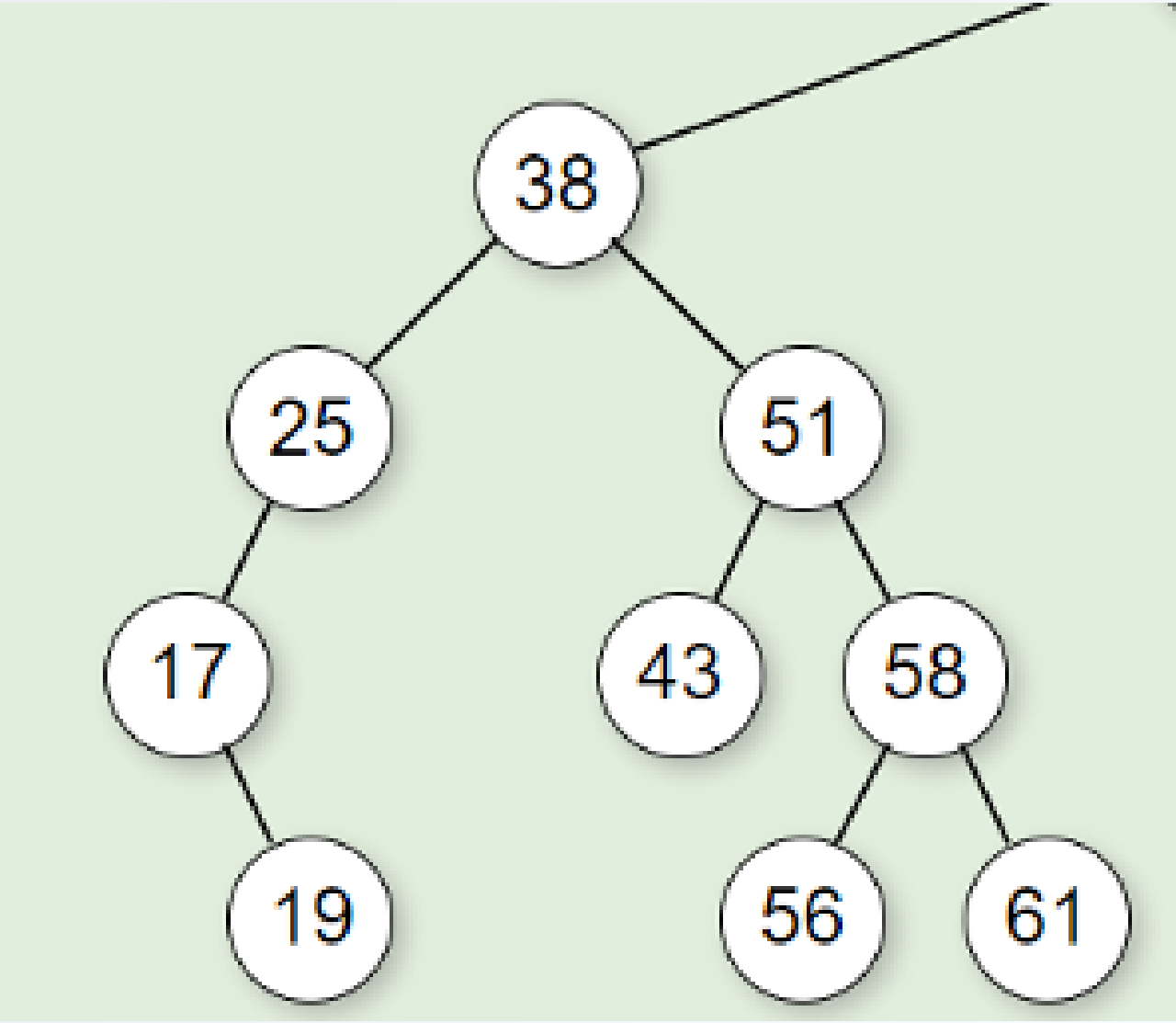


Inorder Traversal

For the tree rooted at 25, the inorder traversal is:

~~Visit the left subtree (rooted at 17),~~ then the current node (25), then the right subtree (none!).

🖨️➡️ 17 19 25

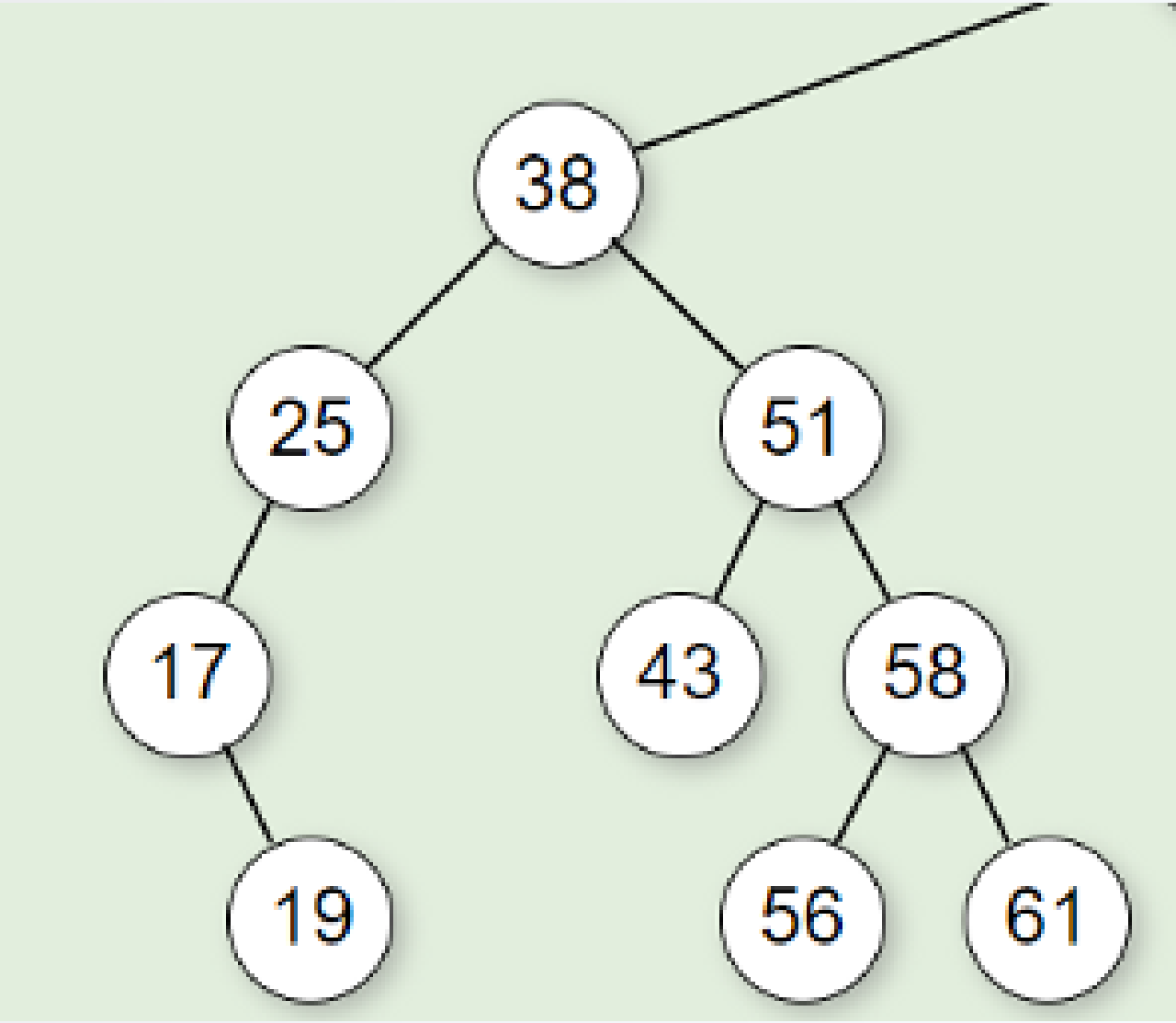


Inorder Traversal

For the tree rooted at 38, the inorder traversal is:

~~Visit the left subtree (rooted at 25),~~ then the current node (38), then the right subtree (rooted at 51).

🖨️➡️ `17 19 25 38`

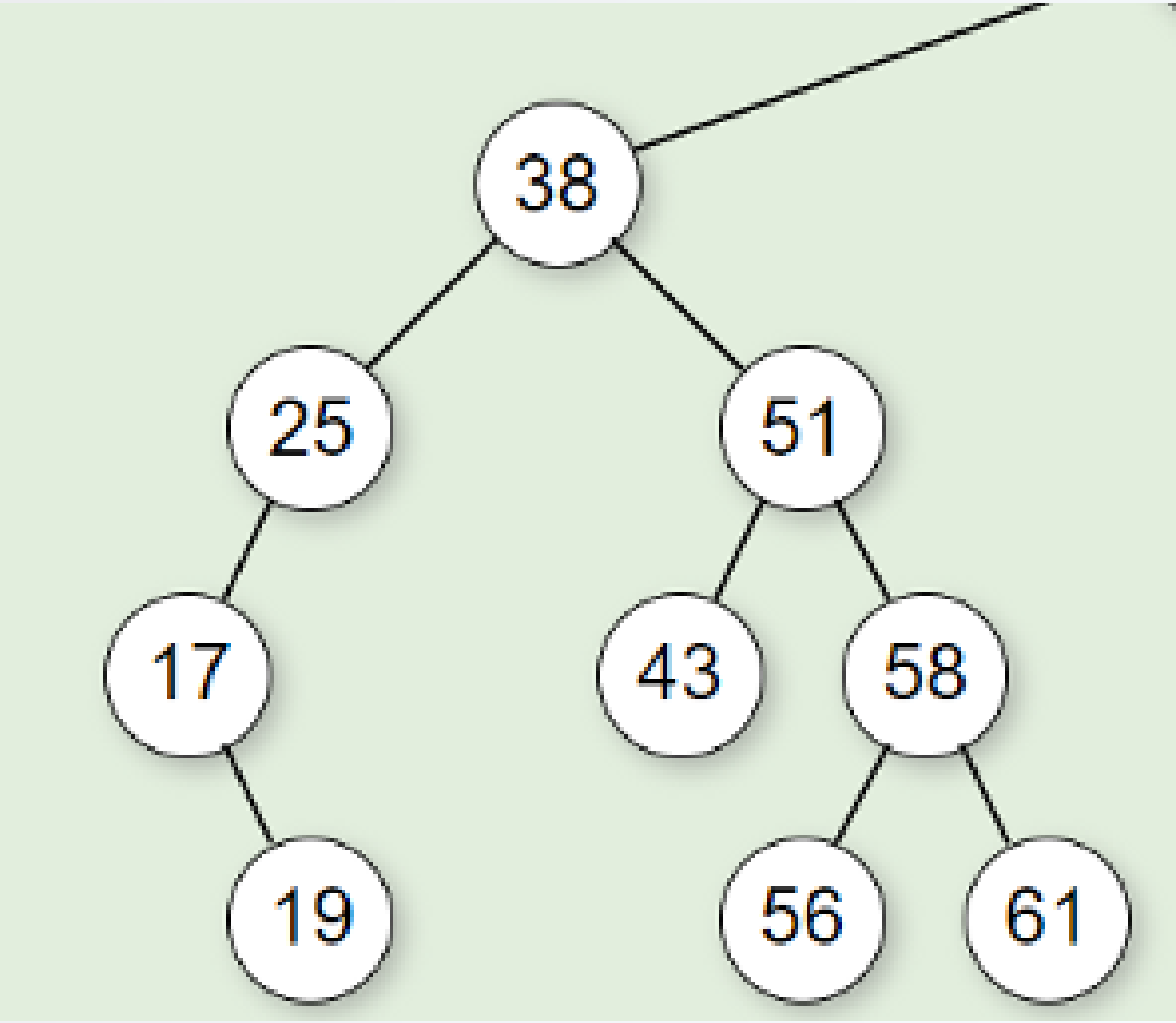


Inorder Traversal

For the tree rooted at 51, the inorder traversal is:

Visit the left subtree (rooted at 43), then the current node (51), then the right subtree (rooted at 58).

🖨️➡️ 17 19 25 38

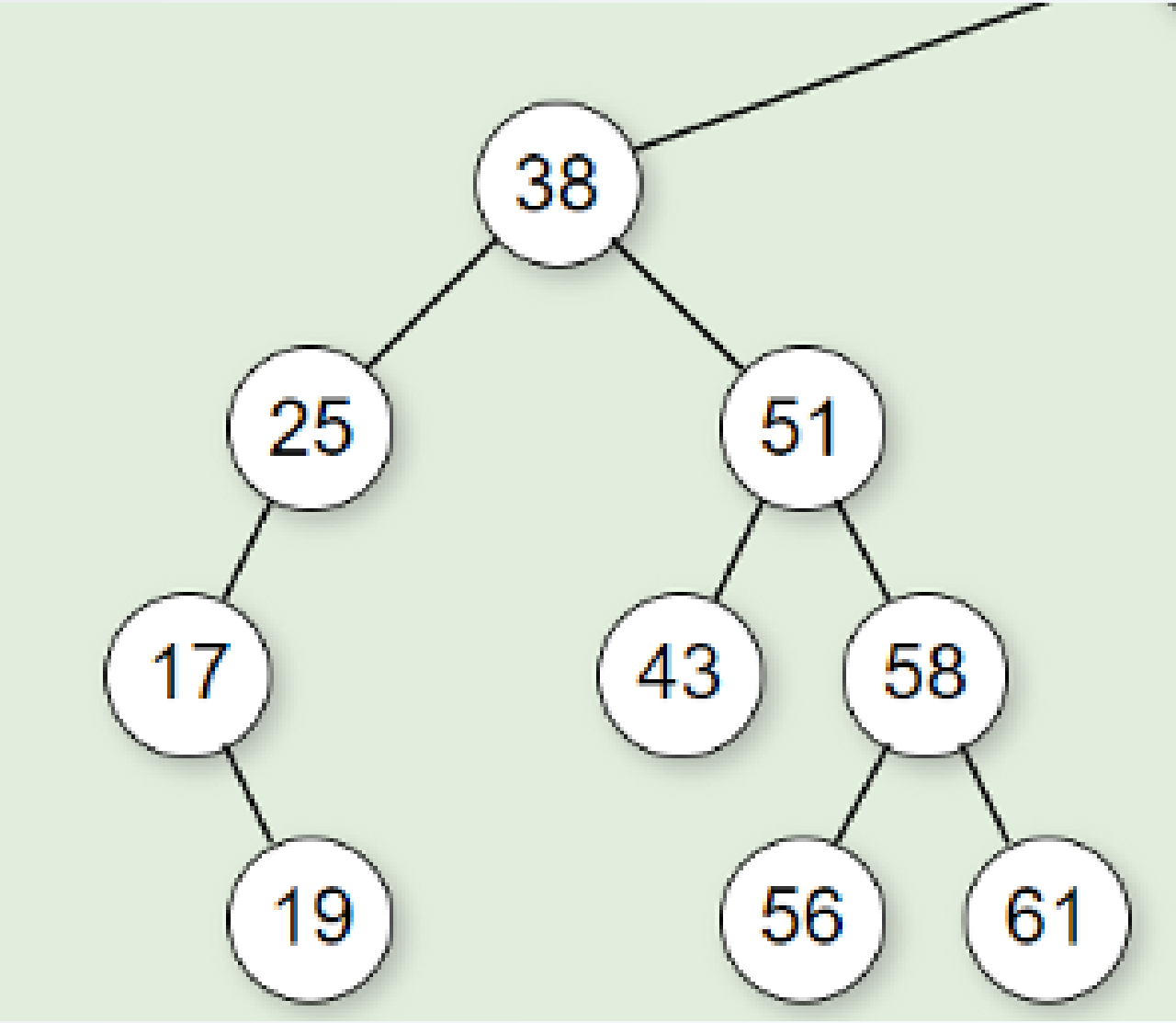


Inorder Traversal

For the tree rooted at 43, the inorder traversal is:

Visit the left subtree (none!), then the current node (43), then the right subtree (none!).

🖨️➡️ 17 19 25 38 43

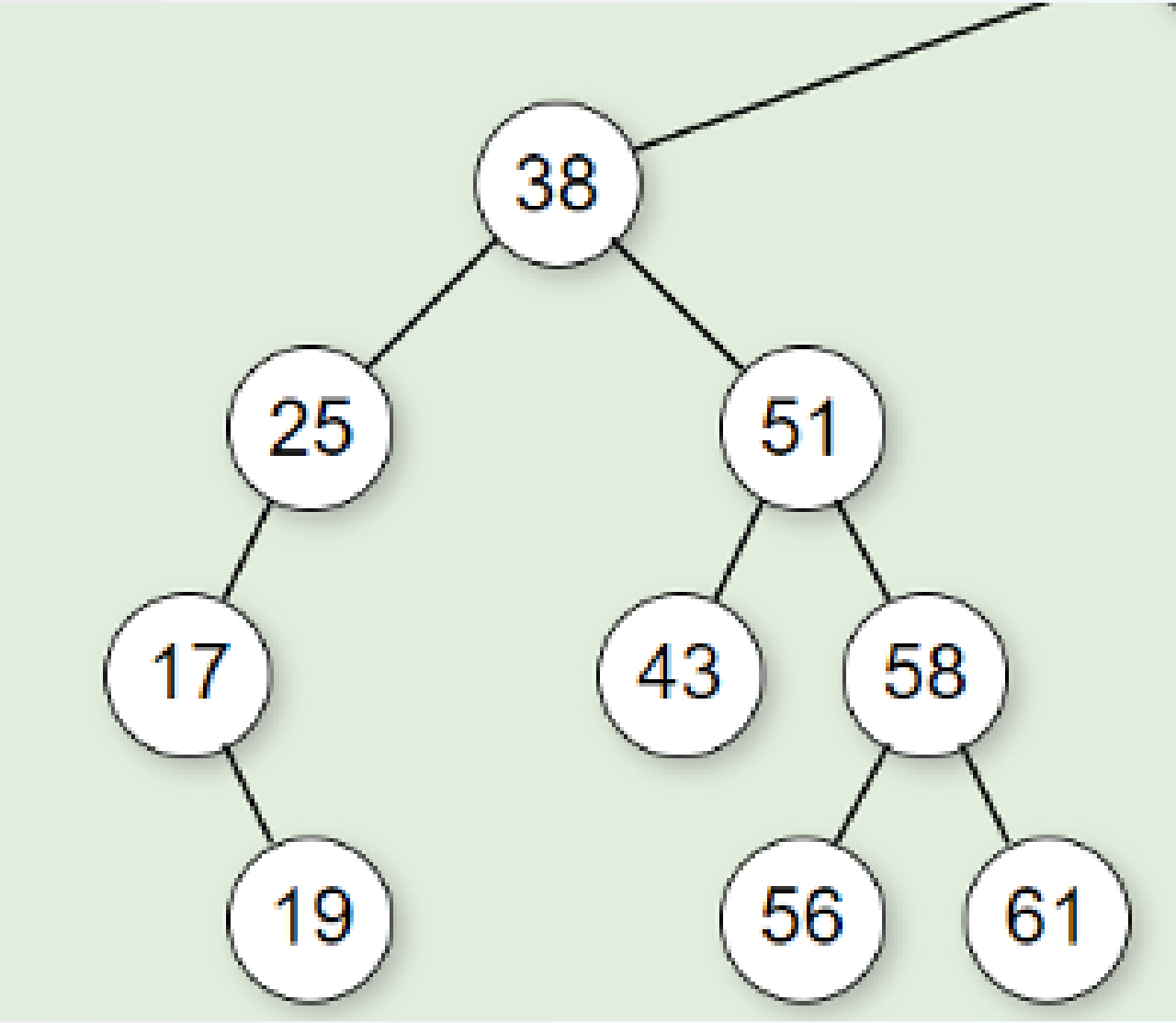


Inorder Traversal

For the tree rooted at 51, the inorder traversal is:

~~Visit the left subtree (rooted at 43),~~ then the current node (51), then the right subtree (rooted at 58).

🖨️➡️ `17 19 25 38 43 51`

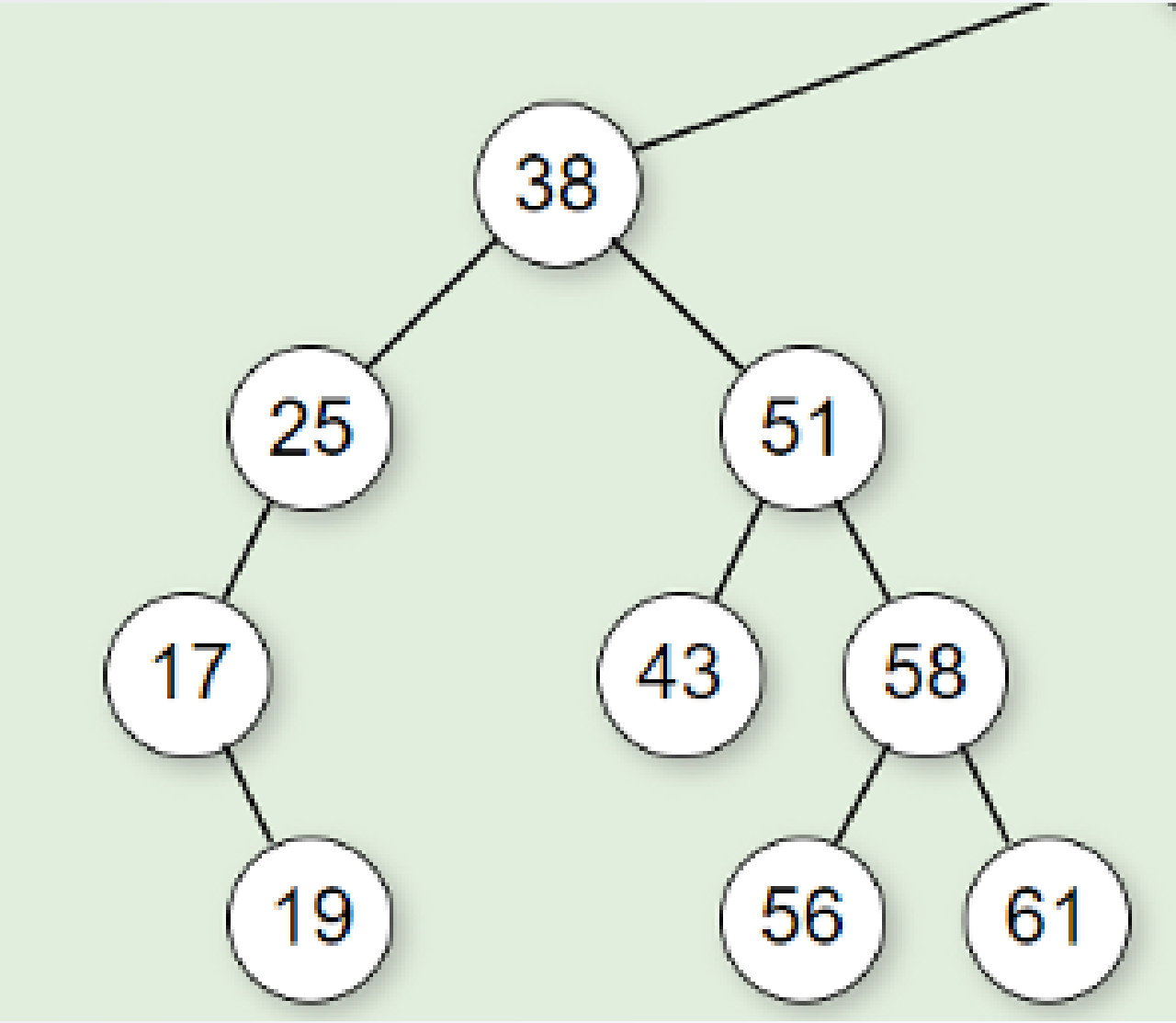


Inorder Traversal

For the tree rooted at 58, the inorder traversal is:

Visit the left subtree (rooted at 56), then the current node (58), then the right subtree (rooted at 61).

🖨️➡️ `17 19 25 38 43 51`

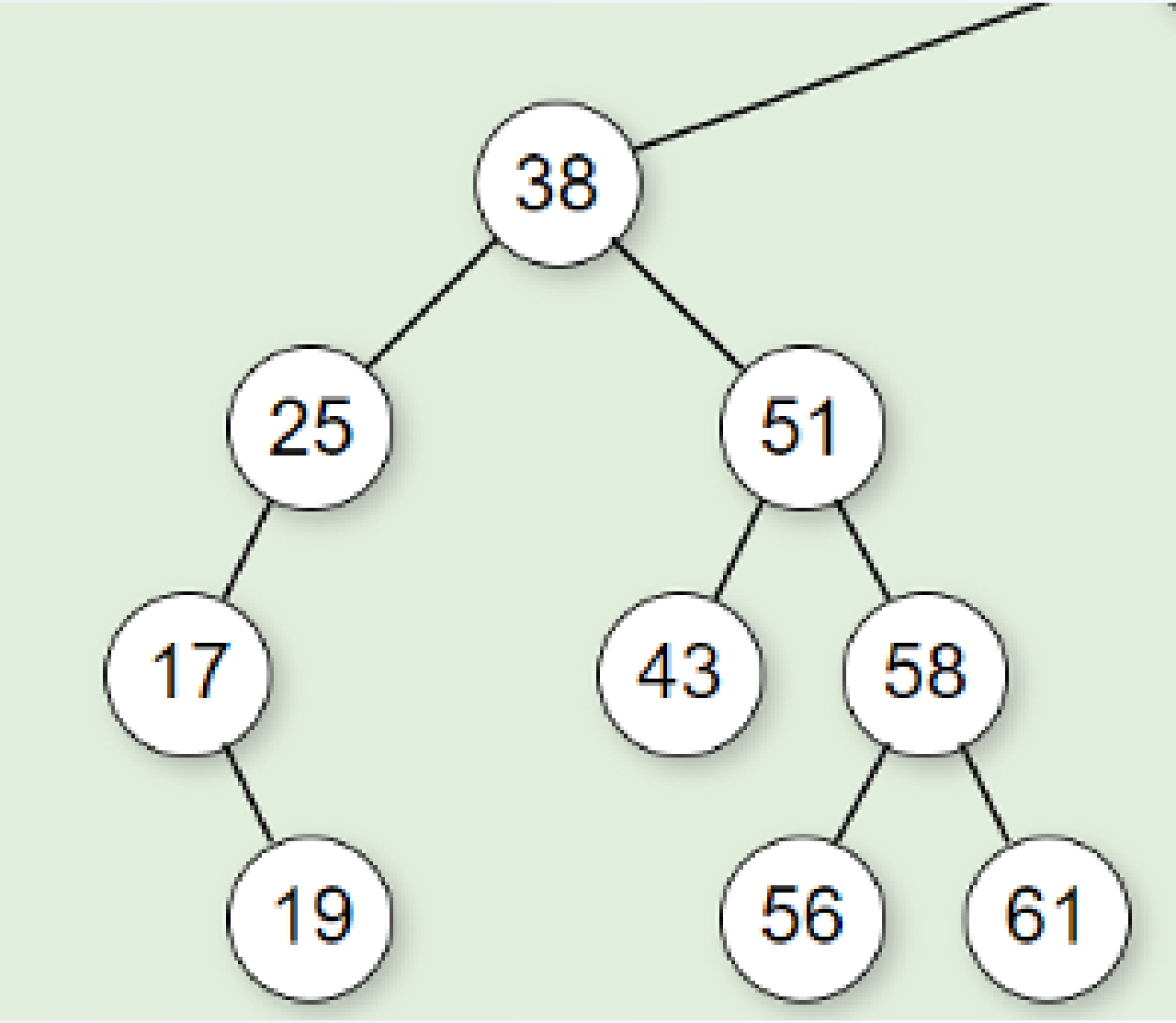


Inorder Traversal

For the tree rooted at 56, the inorder traversal is:

Visit the left subtree (none!), then the current node (56), then the right subtree (none!).

🖨️➡️ `17 19 25 38 43 51 56`

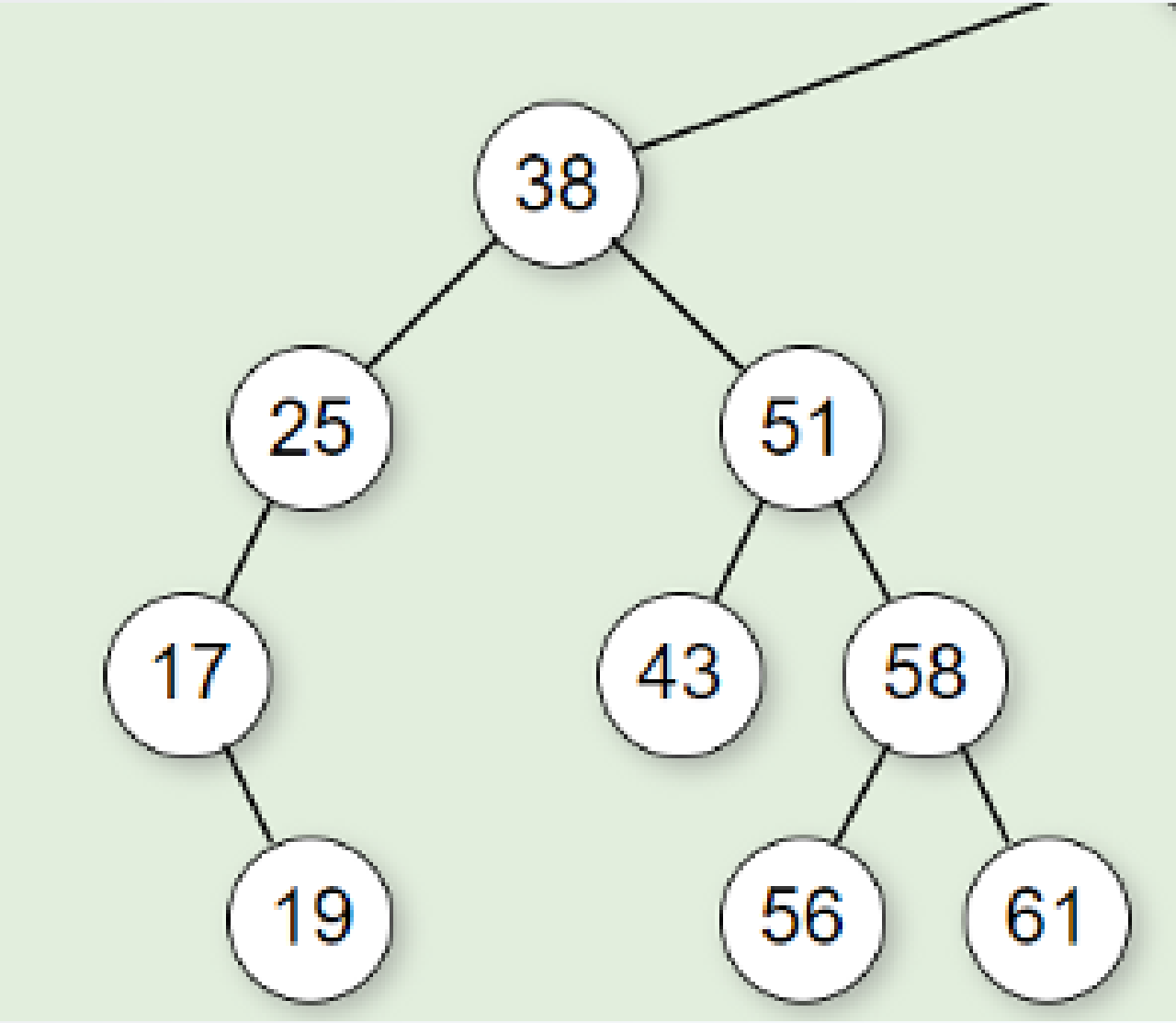


Inorder Traversal

For the tree rooted at 58, the inorder traversal is:

~~Visit the left subtree (rooted at 56),~~ then the current node (58), then the right subtree (rooted at 61).

🖨️➡️ 17 19 25 38 43 51 56 58

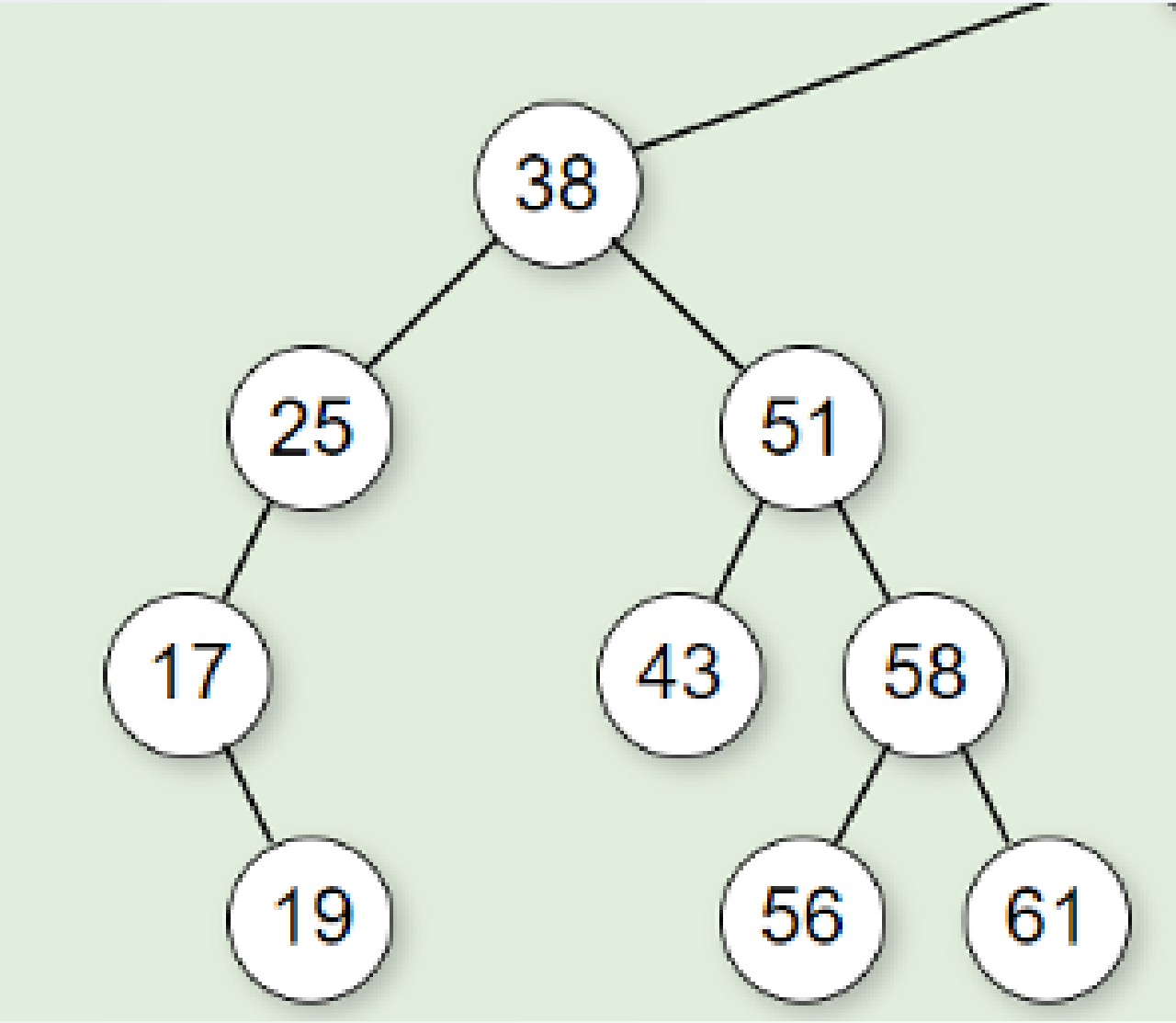


Inorder Traversal

For the tree rooted at 56, the inorder traversal is:

Visit the left subtree (none!), then the current node (61), then the right subtree (none!).

🖨️➡️ 17 19 25 38 43 51 56 58 61



Binary Search Tree

Sorted Data Structure with the operations we want for our Symbol Table:

- add
- remove
- contains

`TreeSet` and `TreeMap` in Java are implemented using a BST. **What are the complexities of these operations?**

BST: implementation

Data fields

- Root of the BST
- Number of nodes

```
private BSTNode<E> root;
private int size;
private Comparator<E> comparator;
```

`E` does not have to be `Comparable`! 🤔

- In order to place an element in a BST, we do need to compare it to other values...
- If `E` is `Comparable`, use the natural ordering
- If `E` is not `Comparable`, the user can provide a Comparator

BSTNode: implementation

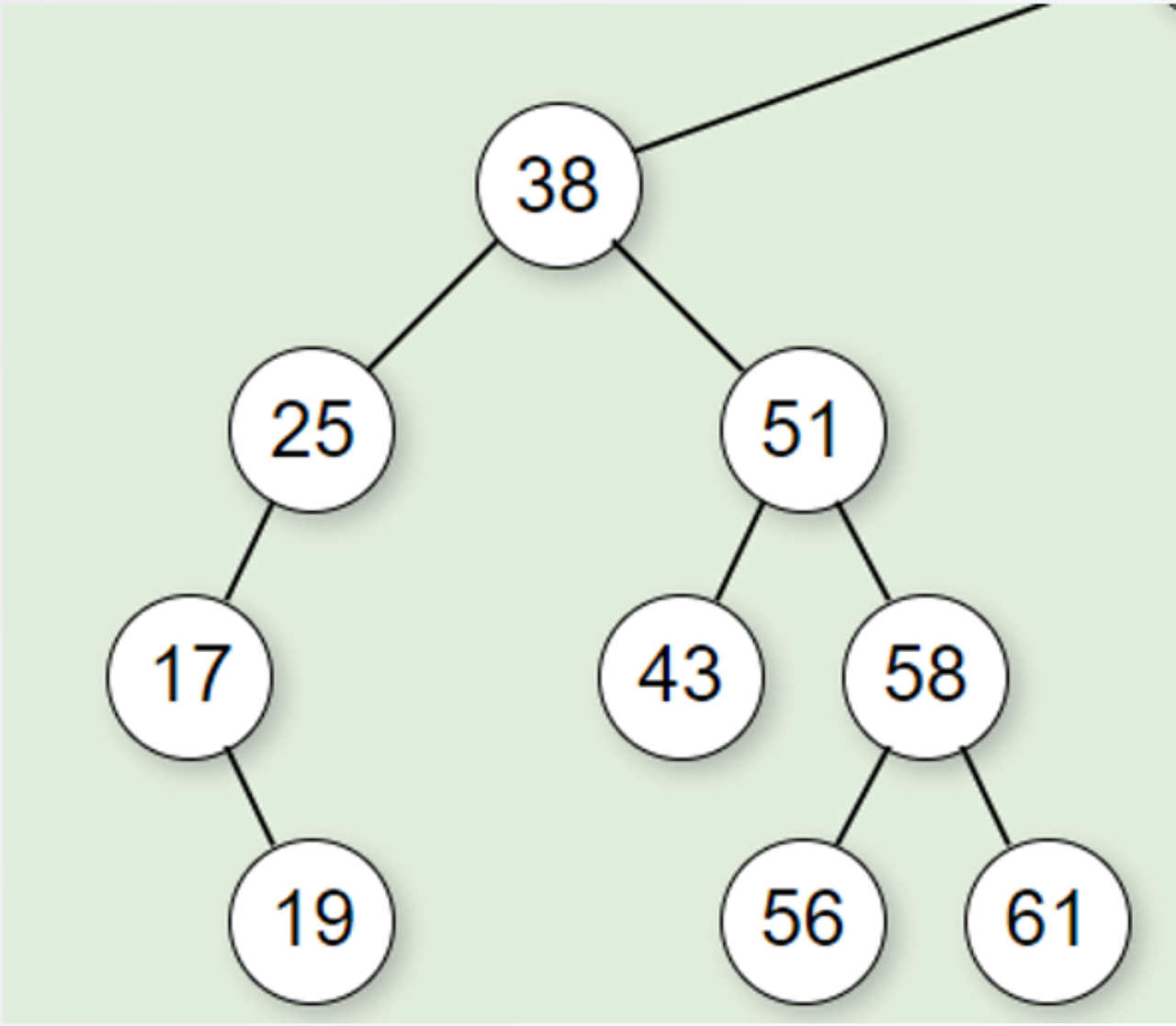
Binary Node instance variables (polymorphic)

- `E element;`
  - Element for this node, stored in BSTLeaf and BST Internal
- `BSTNode left;`
  - Pointer to left child, only stored in Internal
- `BSTNode right;`
  - Pointer to right child, only stored in Internal

BST: contains

High-level description:

- If the tree is null, return false
- Check the root of the tree
  - if we found the element, return true
  - if we don't find the element, recurse on the subtree where we'd expect to find the element



BST: contains

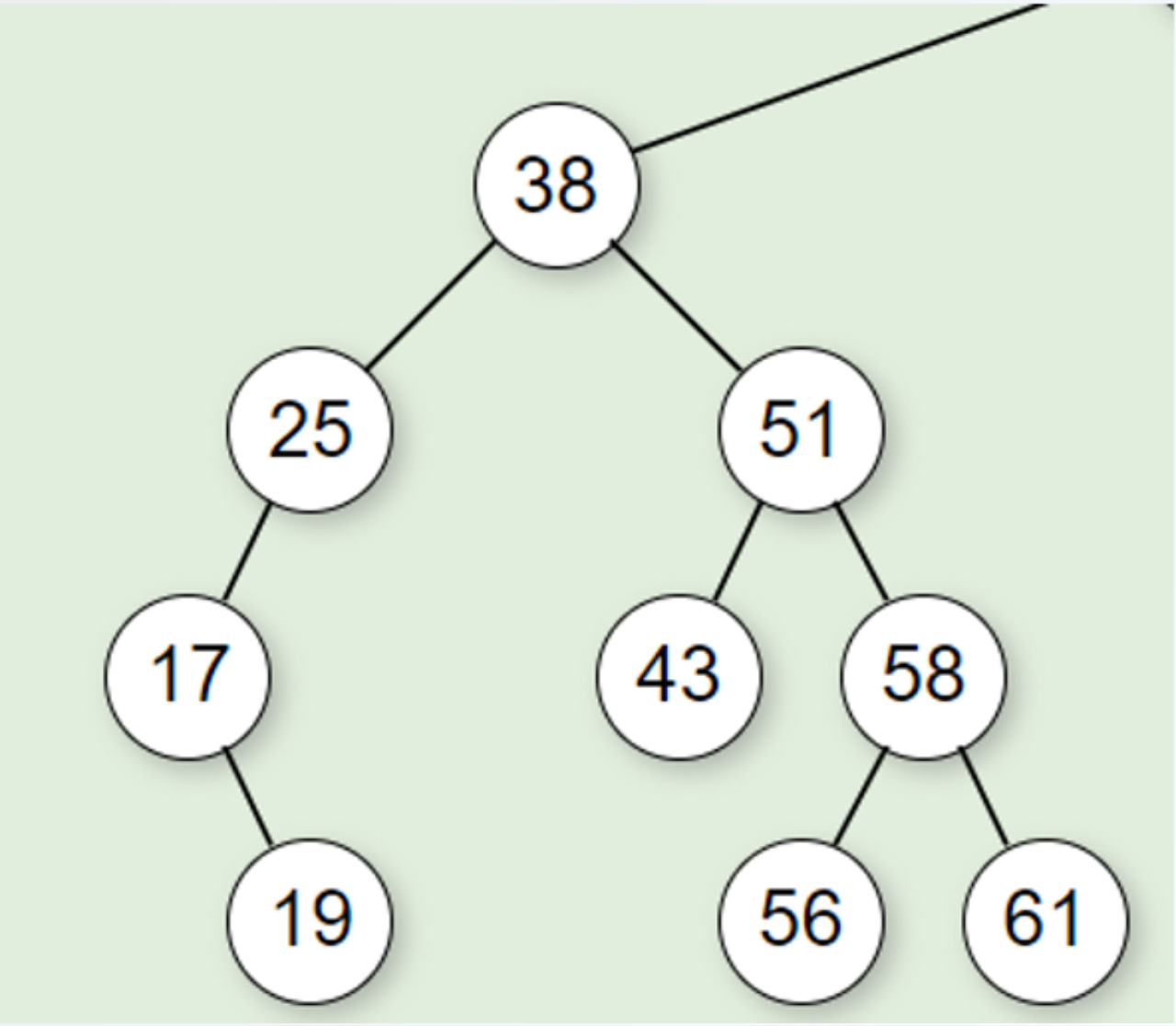
```
private boolean containsHelper(Comparable<E> e, BSTNode<E> treeRoot) {
   if (treeRoot == null) {
           return false;
   }
   int comparison = e.compareTo(treeRoot.getValue());
   boolean isLeaf = treeRoot.isLeaf();
   if (comparison == 0) {
           return true;
   } else if (comparison < 0) {
           // if this is a leaf, STOP! if this is internal, then continue the search
           // to the left subtree
           return !isLeaf && containsHelper(e, ((BSTInternal<E>) treeRoot).getLeft());
   } else {
           // if this is a leaf, STOP! if this is internal, then continue the search
           // to the right subtree
           return !isLeaf && containsHelper(e, ((BSTInternal<E>) treeRoot).getRight());
   }
}
```

BST: contains

Runtime analysis: at most d recursive calls, each of which takes constant time in addition to any recursive calls.

$\Rightarrow O(d)$ runtime.

Here, 9 nodes, but at most 4 recursive calls.

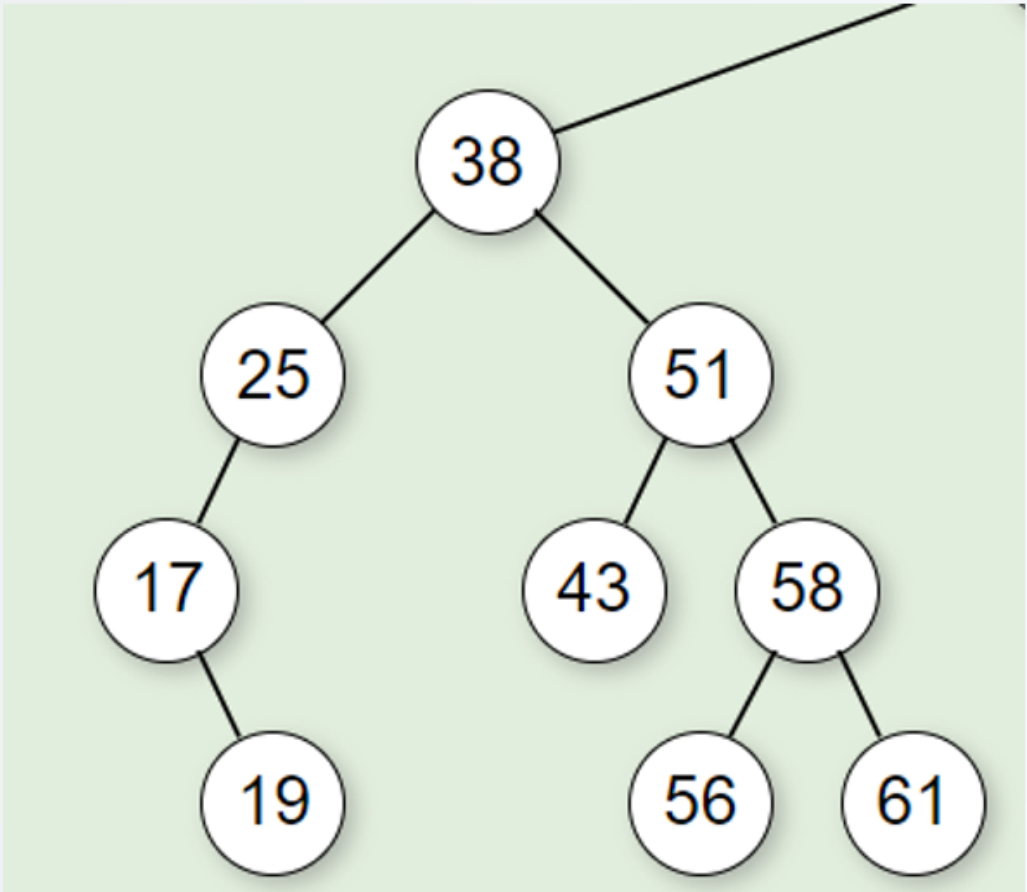


BST: add

High-level description:

- If the tree is null, put the node here
- Check the root of the tree
  - if we found the element, do nothing
  - if we don't find the element, recursively add the element in the left subtree if it's less than the root, otherwise add in the right.

What happens if we insert 18, insert 28, insert 41?



BST: add complexity

- Base case: inserting into an empty (sub)tree of depth 0 is $O(1)$
- Recursive case: Follow a Left or Right child pointer ($O(1)$) and then insert into a subtree with a depth of at most $d - 1$.

Analysis:

- $T(d) = c + T(d - 1)$
- Recurrence relation gives $O(d)$ runtime

What good is O(d)?

Starting from an empty tree, add values 63, 62, 61, ... 3, 2, 1 in that order.

- What is the shape of the tree, and what is its depth in terms of the number of nodes?
- What would be the order of additions that would lead to the least-depth tree?

Main Idea of BST Complexity

- The important operations (add, contains, remove) are all $O(d)$ operations.
- In the worst case, a *pathological* BST of n values is just a linked list where $d \in O(n)$.
- Fortunately, in the best (and average!) case, $d \in O(\log n)$.

In a few weeks, we will study s*elf-balancing tree*s that allow us to guarantee that $d \in O(\log n)$ without even increasing runtime complexity of the operations!

BST: remove

- Find the node (basically **contains**)
- Remove the node
  - Node was a leaf
    - Set reference to the node to null
  - Node had only one child
    - Make the reference to the node to point to its (not null) child
  - Node has two children
    - BST property must be maintained
    - Swap the value stored in the node with the largest value in its left subtree
    - Delete the largest node in the left subtree

BST: remove implementation

- Requires a **parent** pointer in addition to the typical **current** pointer we've been passing around (implicitly with recursion)
- Possible with recursion as well, but the provided implementation is iterative to show you how that looks

Runtime analysis: $O(d)$, since it's a call to `contains` and then a constant number of operations to remove the node.

- Worst case (badly unbalanced BST): $O(n)$
- Average case (balanced tree): $O(\log n)$

Activity

Draw the BST that results when you insert items with keys: 3, 4, 1, 6, 9, 5, 7, 2

Rebuild the above tree after removing 3

TreeSet is a BST

- Implementation of the Set ADT using a BST
- Provides $O(\log n)$ time for `add`, `remove`, and `contains` since they are self-balancing trees.

TreeMap is a BST

The BST, as we've implemented it, is **also** pretty much already a TreeMap!

- We have already rejected duplicates, so that map invariant is maintained
- BST **add**, **remove**, and **contains** behave similar to **put(K key, V value)**, **remove(K key)**, and **containsKey(K key)**
- Problem: The BST stores individual pieces of data, but the TreeMap wants to store **key-value pairs.**
- Solution: Write an *inner-class* of Entry objects and store those in the BST ordered by the keys.

Back to the Symbol Table

...now we have an efficient data structure for `add`, `remove`, and `contains` operations on `Symbol` records.

- Let's have our `SymbolTable` be a `TreeSet<Symbol>`
- We can use the `add`, `remove`, and `contains` methods of `TreeSet` to implement the `add`, `remove`, and `contains` methods of `SymbolTable`.

Test Cases

Consider what should happen for each of these cases.

- Adding a new symbol to an empty table
- Adding a new symbol to a non-empty table
- Adding a symbol that is already in the table
- Checking for a symbol that is present in the table
- Checking for a symbol that is not present in the table
- Removing a symbol that is present in the table
- Removing a symbol that is not present in the table

Test Cases

(✅ for acceptable/ ❌ for unacceptable)

- Adding a new symbol to an empty table ✅
- Adding a new symbol to a non-empty table ✅
- Adding a symbol that is already in the table (❌, redeclaring a variable!)
- Checking for a symbol that is present in the table ✅
- Checking for a symbol that is not present in the table ✅
- Removing a symbol that is present in the table ✅
- Removing a symbol that is not present in the table (❌, shouldn't happen)

Writing the Code

(Check the `.zip` file on the website for today's class.)