Pre-Class Work 13: Binary Trees

Trees [Background Reading]

🌲 Motivation

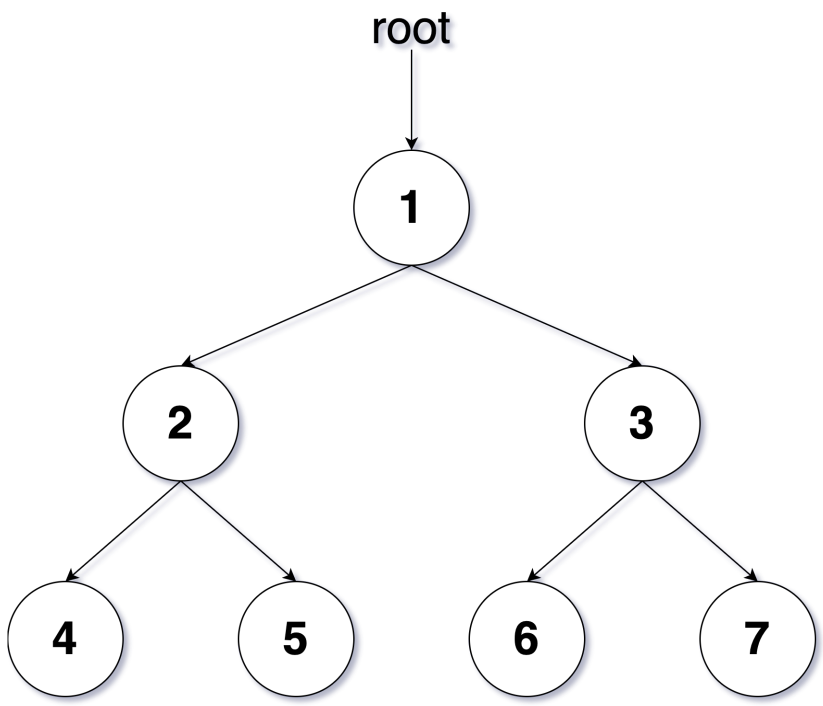
Trees are a very common type of data structure used in computer science to represent a myriad of things. For example:

- A tree is what's used to implement `TreeSet` and `TreeMap` to keep data sorted and efficiently findable
- Folders and files in a computer form a tree where there is a top folder (the root) and sub-files (children). We'll see that just like the file was defined recursively, a tree will be defined recursively as well.
- A family tree (parents and children) to track genealogy or an organizational chart.
- And much much more!

🌲 Binary Trees

Binary trees are one example of the tree data structure. As the name suggests, each node in a **binary tree** has at most two children, which are referred to as the left child and the right child. This is similar to the idea of the `ListNode` forming up a linked list, where a linked list node stores a "next" reference.

You can think of **binary tree** nodes as storing two "next" references! The top-most node in the tree is called the **root** node, while the nodes that do not have any children are called **leaves**.

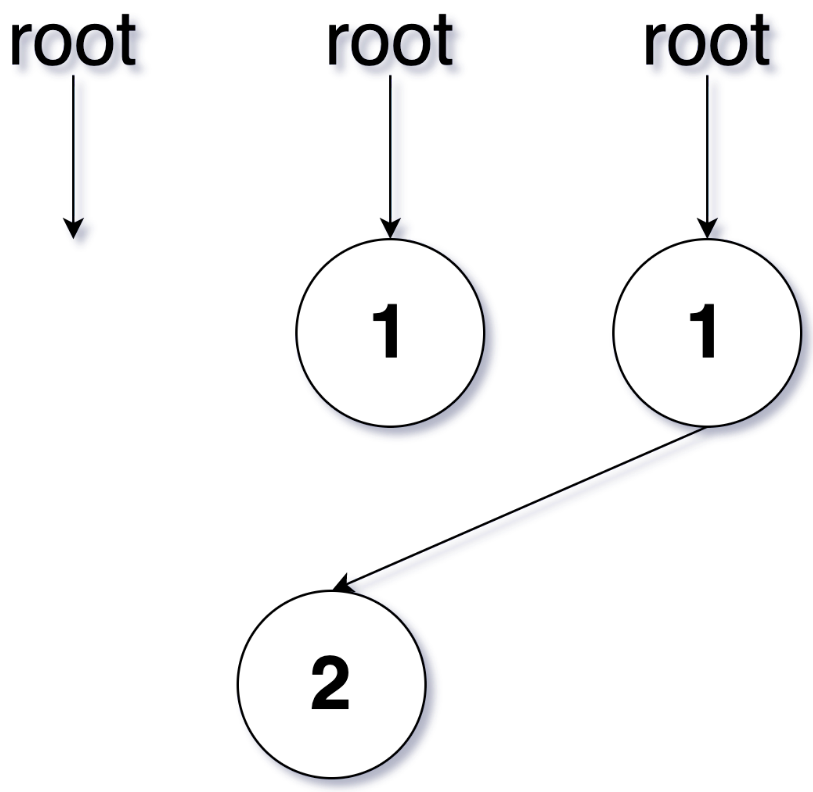


The binary tree has a very natural recursive definition that matches a tree of any height! A tree is one of two possibilities

- An empty tree (or `null`)
- A **root** node that contains
    - **data** (e.g. an `int`)
    - a **left** subtree
    - a **right** subtree

These left/right subtrees also follow the definition of trees above! This means they could be empty (`null`), or they could be a node with some data and two subtree children! This definition could go on and on for as long as necessary since it is recursive.

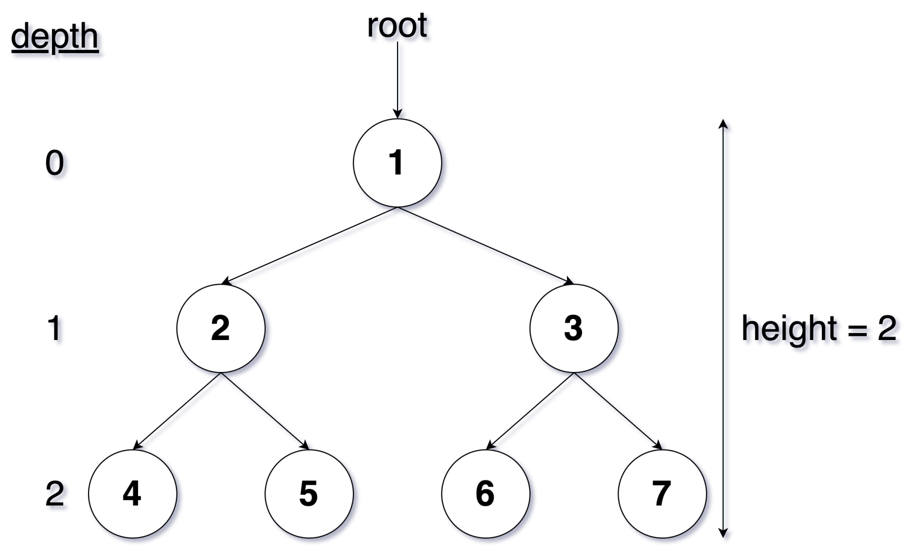
Here are some examples of binary trees (notice they all match this recursive definition since some of the subtrees are empty which is allowed!).



🔑 Terminology

There is usually a lot of terminology that goes along when learning trees. Here are the most common terms you will hear being used when discussing trees:

- A **node** is a building block the tree that will contain some data value and its left/right subtrees.
    - **root**: The topmost node of a tree
    - **leaf**: A node that has no children (i.e. both children are empty trees)
    - **branch**: An internal node; neither the root nor a leaf.
    - **parent** of a node: A node that refers to this node (in the picture below, 1 is a parent of both 2 and 3)
    - **child** of a node: A node that is referred to by this node (in the picture below, 2 and 3 are children of 1)
    - **sibling** of a node: Another node in the tree with a common parent (2 and 3 are siblings in the picture below).
- **subtree**: The smaller tree of nodes to the left/right of the current node
- **height**: The length of the longest path from the root to any node (the height of the tree in the picture below is 2)
- **level** or **depth**: The length of the path from a root to a given node.



􏰀 Main Points

- Trees are a very common construct in computer science. A **binary tree** is a variant of the tree structure where each node has at most two children (the left and right child)
  - We can think of binary trees as storing two "next" references!
- The top-most node in a binary tree is the **root**, and nodes that do not have any children are **leaves**.
- A binary tree has a naturally **recursive** definition and is one of two possibilities:
  - An empty tree (or `null`)
  - A **root** node that contains
    - **data** (e.g. an `int`)
    - a **left** subtree
    - a **right** subtree
- Common terms we will see when working with binary trees include **node**, **root**, **leaf**, **branch**, **parent**, **child**, **sibling**, **subtree**, **height**, and **level**/**depth**.

Anatomy of Binary Trees [Background Reading]

The best way to learn about a binary tree is to implement our own version of one! We will focus on making a tree that stores `int` values and call this class an `IntTree`.

`IntTreeNode`

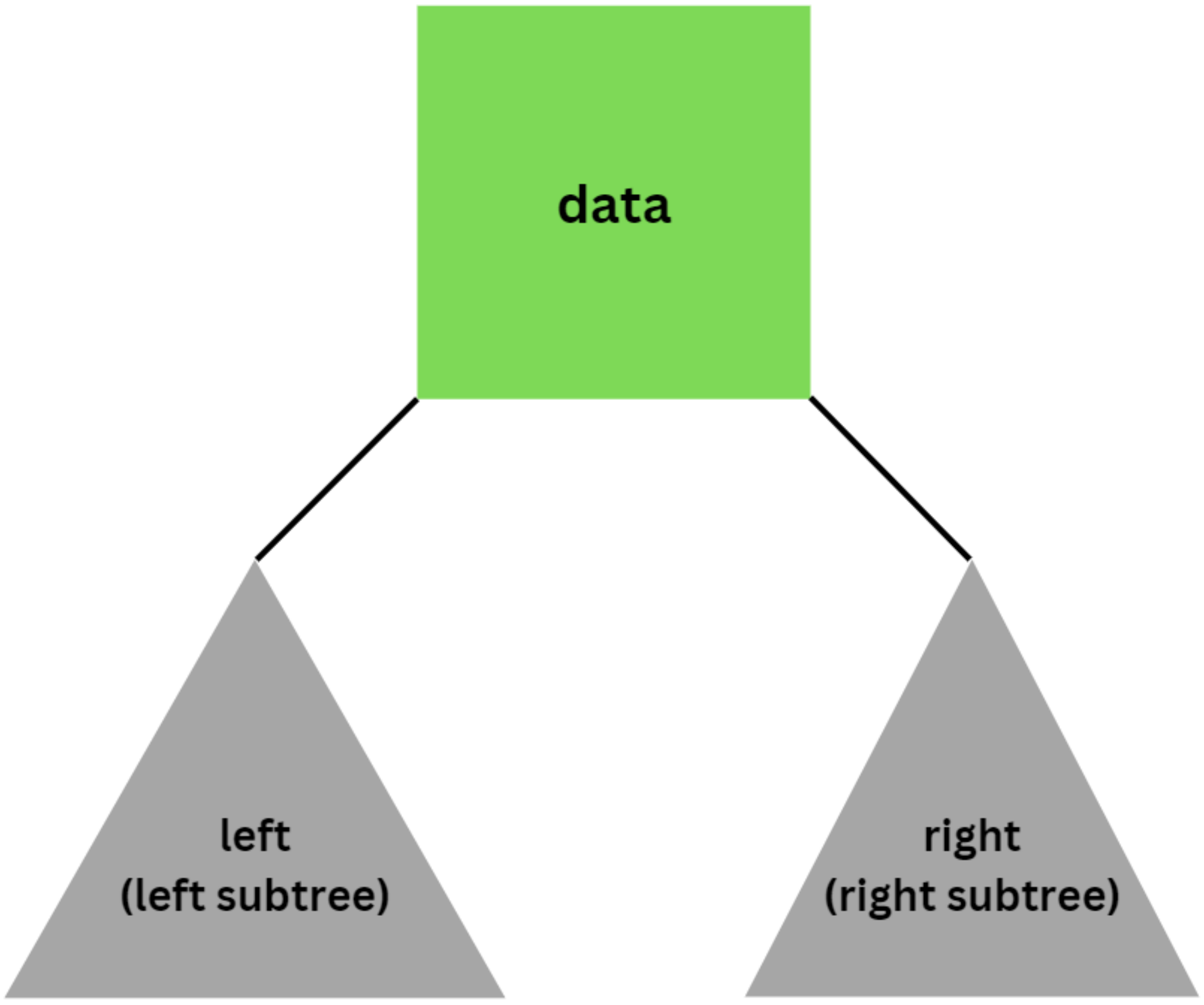
Before we can implement `IntTree`, we will need to consider what the nodes will keep track! Recall that **binary tree** nodes are the individual building blocks in a **binary tree** similar to how a list node is a building block for a linked list. We will call our **binary tree** node `IntTreeNode` and implement it as a `static` class inside `IntTree`.

As a technical note that's reason is outside the scope of CSE 123, we are making `IntTreeNode` `static` since this more appropriately fits our needs. You don't need to understand why it is `static`, all you need to know is that you define the node class inside the tree class.

```
public class IntTree {
   // TODO: Implement IntTree

   public static class IntTreeNode {
        public int data; // data stored at this node
        public IntTreeNode left; // reference to left subtree
        public IntTreeNode right; // reference to right subtree
    }
}
```

Here, `data` represents the value stored inside the `IntTreeNode`. `left` and `right` are references to the next `IntTreeNode`! You can think of this as having two "next" references in a `ListNode`. We'll need to provide an implementation for the constructors so that `IntTree` can actually construct the nodes:

```
    public static class IntTreeNode {
        public int data; // data stored at this node
        public IntTreeNode left; // reference to left subtree
        public IntTreeNode right; // reference to right subtree

        // Constructs a leaf node with the given data.
        public IntTreeNode(int data) {
            this(data, null, null);
        }

       // Constructs a leaf or branch node with the given data and links.
       public IntTreeNode(int data, IntTreeNode left, IntTreeNode right) {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }
```



`IntTree`

Now that we finished implementing `IntTreeNode`, we can implement `IntTree`. First, let's consider what fields we'll need. Similar to how `LinkedIntList` was implemented, we only need to keep track of one node in order to have access to the entire tree.

In `LinkedIntList`, that field which gave us access to every other node was called `front`. We will call our field `overallRoot`:

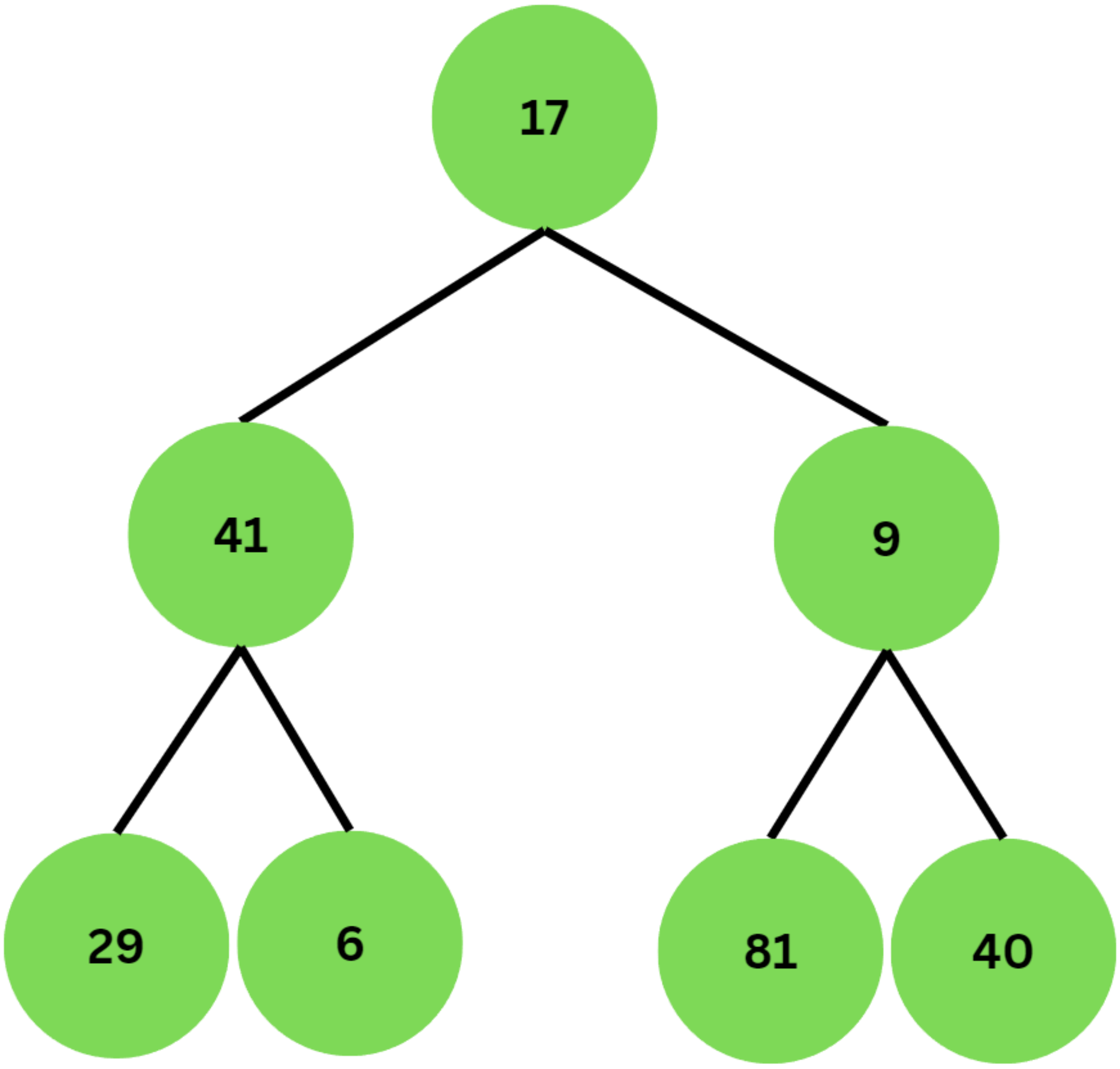
```
// This class represents a tree of integers
public class IntTree {
    private IntTreeNode overallRoot;

    // Constructs an empty tree
    public IntTree() {
        overallRoot = null;
    }

    // Class that represents a single node in the tree
    private static class IntTreeNode {
        public int data; // data stored at this node
        public IntTreeNode left; // reference to left subtree
        public IntTreeNode right; // reference to right subtree

        // Constructs a leaf node with the given data.
        public IntTreeNode(int data) {
            this(data, null, null);
        }

        // Constructs a leaf or branch node with the given data and links.
        public IntTreeNode(int data, IntTreeNode left, IntTreeNode right) {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }
}
```

For now, we will modify our constructor to create a default tree (we'll discuss a better way to add and remove nodes in a future lesson):

```
 // This class represents a tree of integers
public class IntTree {
    private IntTreeNode overallRoot;

    // Constructs a tree with default
    public IntTree() {
        overallRoot = new IntTreeNode(17);
        overallRoot.left = new IntTreeNode(41);
        overallRoot.right = new IntTreeNode(9);
        overallRoot.left.left = new IntTreeNode(29);
        overallRoot.left.right = new IntTreeNode(6);
        overallRoot.right.left = new IntTreeNode(81);
        overallRoot.right.right = new IntTreeNode(40);
    }

    // Class that represents a single node in the tree
    private static class IntTreeNode {
        public int data; // data stored at this node
        public IntTreeNode left; // reference to left subtree
        public IntTreeNode right; // reference to right subtree
        // Constructs a leaf node with the given data.
        public IntTreeNode(int data) {
            this(data, null, null);
        }

        // Constructs a leaf or branch node with the given data and links.
        public IntTreeNode(int data, IntTreeNode left, IntTreeNode right) {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }
}
```

As a word of caution, what we are doing in the constructor is not great code quality. You are welcome to write similar code as a method of testing but you should not submit code which chains multiple accesses. We'll cover a better way to initialize trees in a future lesson.

Here is what `IntTree` is initialized to:



As we can see, the root node has a value of 17. Its left node has a value of 41 and the left node's children are 29 and 6. The root node's right node has a value of 9 and its children are 81 and 40.

🔑 Main Points

- We created our own version of a binary tree by implementing `IntTree`, a tree that stores `int` values.
- **Binary tree** nodes are the individual building blocks in a **binary tree** similar to how a list node is a building block for a linked list.
  - We made our own **binary tree** node `IntTreeNode` and implemented it as a `static` class inside `IntTree`.

- We chose to include three fields inside `IntTreeNode`: `data` (the data stored inside the node), `left` (a reference to our left subtree), and `right` (a reference to our right subtree)
- Since we only need to keep track of one node in order to have access to the entire tree, we added one field to our `IntTree` class named `overallRoot`, which is an `IntTreeNode`.
- We then modified our constructor to make a default tree with our final constructor being:

```java
public IntTree() {
        overallRoot = new IntTreeNode(17);
        overallRoot.left = new IntTreeNode(41);
        overallRoot.right = new IntTreeNode(9);
        overallRoot.left.left = new IntTreeNode(29);
        overallRoot.left.right = new IntTreeNode(6);
        overallRoot.right.left = new IntTreeNode(81);
        overallRoot.right.right = new IntTreeNode(40);
}
```

Implementing contains() [Background Reading]

Now that we have the `IntTree` class set up, all that is left is to implement some methods! A common functionality for objects of binary tree classes and other classes that contain data is to see whether or not they contain a certain value.

Let’s give this functionality to objects of our `IntTree` class by implementing `contains()`, which will return a `boolean` value. More specifically, it will return true given `int` value is in our `IntTree` and false otherwise.

We start with the method stub

```
// post: returns true if the given integer is in the IntTree
// returns false otherwise
public boolean contains(int value) {
    // TODO: implement this method
}
```

Private Helper Methods

Recall that binary trees are a recursively defined data structure, meaning it's natural to use recursion to solve binary tree problems. Remember, when we write recursive methods we want each method call to represent one small part of solving the larger problem.

In this case, we should have each method call represent one instance of the definition of our binary tree. That is, it should either deal with the case where we are at the *empty tree*, or it should deal with the case where we are at a *node with potentially left and right subtrees*.

In order to do this, we need some way to keep track of where we currently are in the tree at the time the recursive method is called.

Since we have no way of doing this with just the `public` method, we introduce a `private` helper method that takes in an `IntTreeNode` parameter that represents the current node we are at (this is exactly like what we did in the `print` method in class).

```
// post: returns true if the given integer is in the IntTre
// returns false otherwise
public boolean contains(int value) {
    // TODO: implement this method
}

// post: return true if the tree starting at the given
// IntTreeNode contains the given value. Returns false otherwise
private boolean contains(IntTreeNode root, int value) {
    // TODO: implement this helper method
}
```

Now we can begin to write our recursive method. Let's start with the `private` helper, which will be doing the bulk of the work. Remember, recursive methods have two parts, a base case and a recursive case. We can use the recursive definition of a binary tree to help us spot what our base case should be and what our recursive case should be.

Let's start with the base case.

1▯ Base Case

Remember, we want the base case to be the simplest, most basic version of the problem at hand that we can solve almost immediately. What's a simple tree for which we can immediately tell if it contains our value?

The *empty tree* is simple since we know that the empty tree can't contain our value. How do we know if a tree is empty? If it is `null` then we know there are no nodes in our tree, and thus it is empty. We can then signify that it doesn't contain our value by returning `false`.

The following is a good base case

```java
private boolean contains(IntTreeNode root, int value) {
    if (root == null) {
        return false;
    } else {
        // TODO: implement recursive case
    }
}
```

2▯ Recursive Case

Let's move on to the recursive case, which is when we have a binary tree that is *a node with potentially left and right subtrees*. When writing recursive cases for binary trees, we need to address each part of the definition. As a general rule of thumb, here is a list of cases to consider when working with binary trees:

- Handle current node
- Handle left subtree
- Handle right subtree

We'll also need to figure out somehow link above parts into a larger result. In our `contains()` example, we can handle the current node by seeing if it contains the value (i.e. `root.data ==`

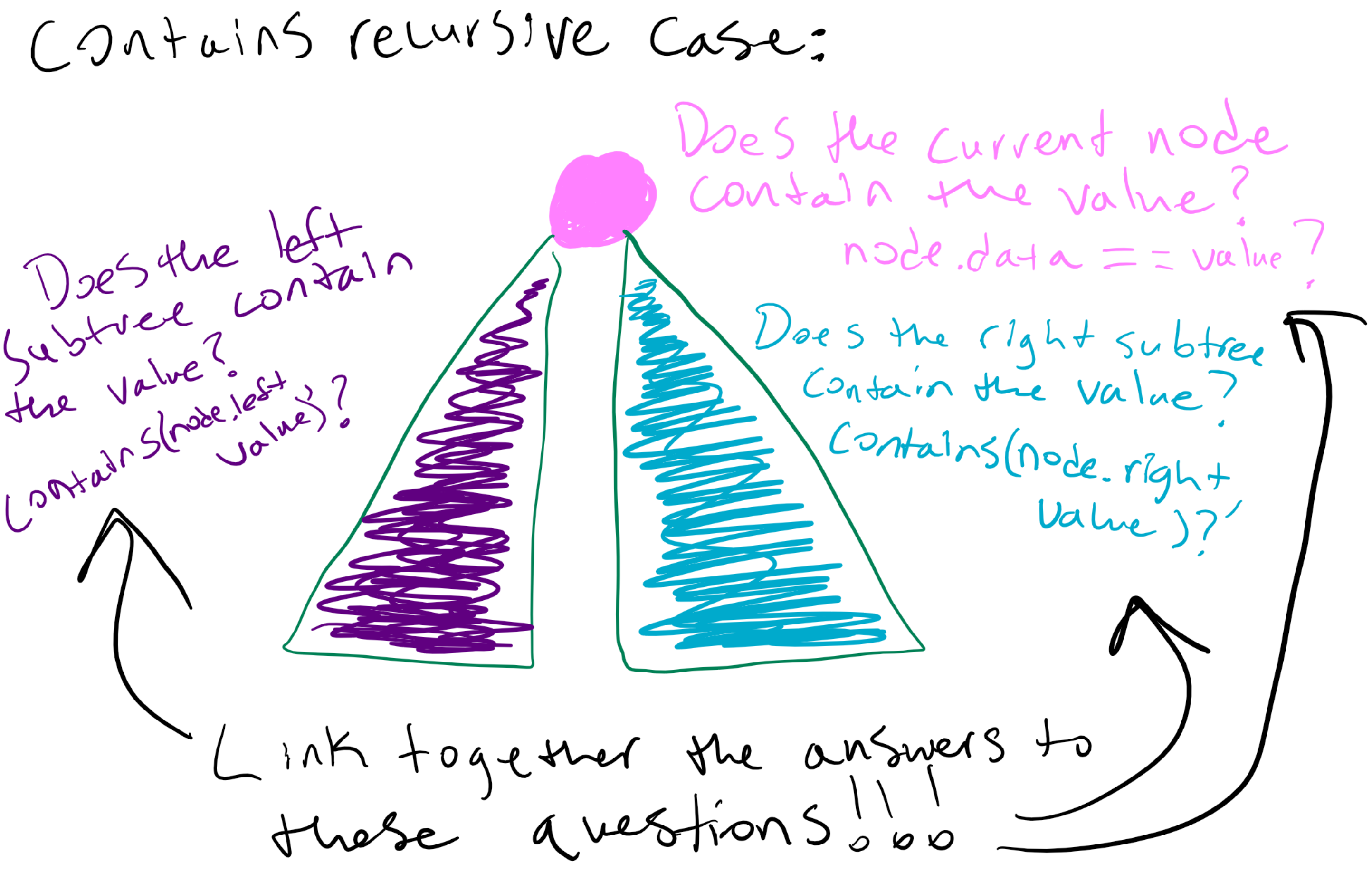
`value`), in which case we want to return `true` to signify that we found the value!

Notice in this case we don't have to recurse because we immediately know the answer. This case checks off the "handle current node" task of our recursive todo list. Our updated method:

```java
private boolean contains(IntTreeNode root, int value) {
    if (root == null) {
        return false;
    } else if (root.data == value) {
        return true;
    } else {
        // TODO: implement recursive case
    }
}
```

Now we need to handle the left and right subtrees. We know that a subtree is really just a binary tree that starts at either `root.left` or `root.right`, so we can see if the value is in a subtree by calling our recursive method on that subtree. This would be a call to `contains(root.left, value)` for the left subtree and `contains(root.right, value)` for the right one.

Here is a visualization:


Contains recursive case:
Does the current node contain the value?
node.data == value?
Does the left subtree contain the value?
contains(node.left, value)?
Does the right subtree contain the value?
contains(node.right, value)?
Link together the answers to these questions!!!

How should we link the two recursive calls on the subtrees together? Sometimes it can help to describe what we want to return in order to figure out how to link the recursive calls together. If the current node does not contain the value (this means we are in the `else` branch), then we want to return `true` if the left subtree **or** the right subtree contains the value. T

hus we can use the logical operator `||` to complete our `private` helper method:

```
private boolean contains(IntTreeNode root, int value) {
    if (root == null) {
        return false;
    } else if (root.data == value) {
        return true;
    } else {
        return contains(root.left, value) || contains(root.right, value);
    }
}
```

Now all that is left is to pair it with the `public` method! We want to start at `overallRoot`, so the completed pair looks as follows:

```
public boolean contains(int value) {
    return contains(overallRoot, value);
}

private boolean contains(IntTreeNode root, int value) {
    if (root == null) {
        return false;
    } else if (root.data == value) {
        return true;
    } else {
        return contains(root.left, value) || contains(root.right, value);
    }
}
```

The most important thing to takeaway from this reading is that we use recursion to solve many binary tree problems because their **recursive definition** lends itself to more succinct and readable recursive solutions.

It’s also important to note that we can use the recursive definition of a binary tree to help guide us when implementing these recursive solutions.

Main Points

- The `contains()` method in **binary trees** will return whether a value is found in the tree (true or false).
- We often use a private **helper method** with additional parameters (such as the current node of the tree) when implementing the `contains()` method and the helper method will perform the recursive search.
- The **base case** in the `contains()` method is when the current node is `null` (empty subtree).

    - The `contains()` method will return **false** in this case since an empty subtree by default contains no values.
- In the **recursive case**, the method checks if the current node contains the value.
    - If it does, the method returns **true**.
    - If it doesn't, the method makes recursive calls on the left and right subtrees and combines their results with the logical OR (`||`) operator.
- We use recursion to solve many binary tree problems because their **recursive definition** lends itself to more succinct and readable recursive solutions.

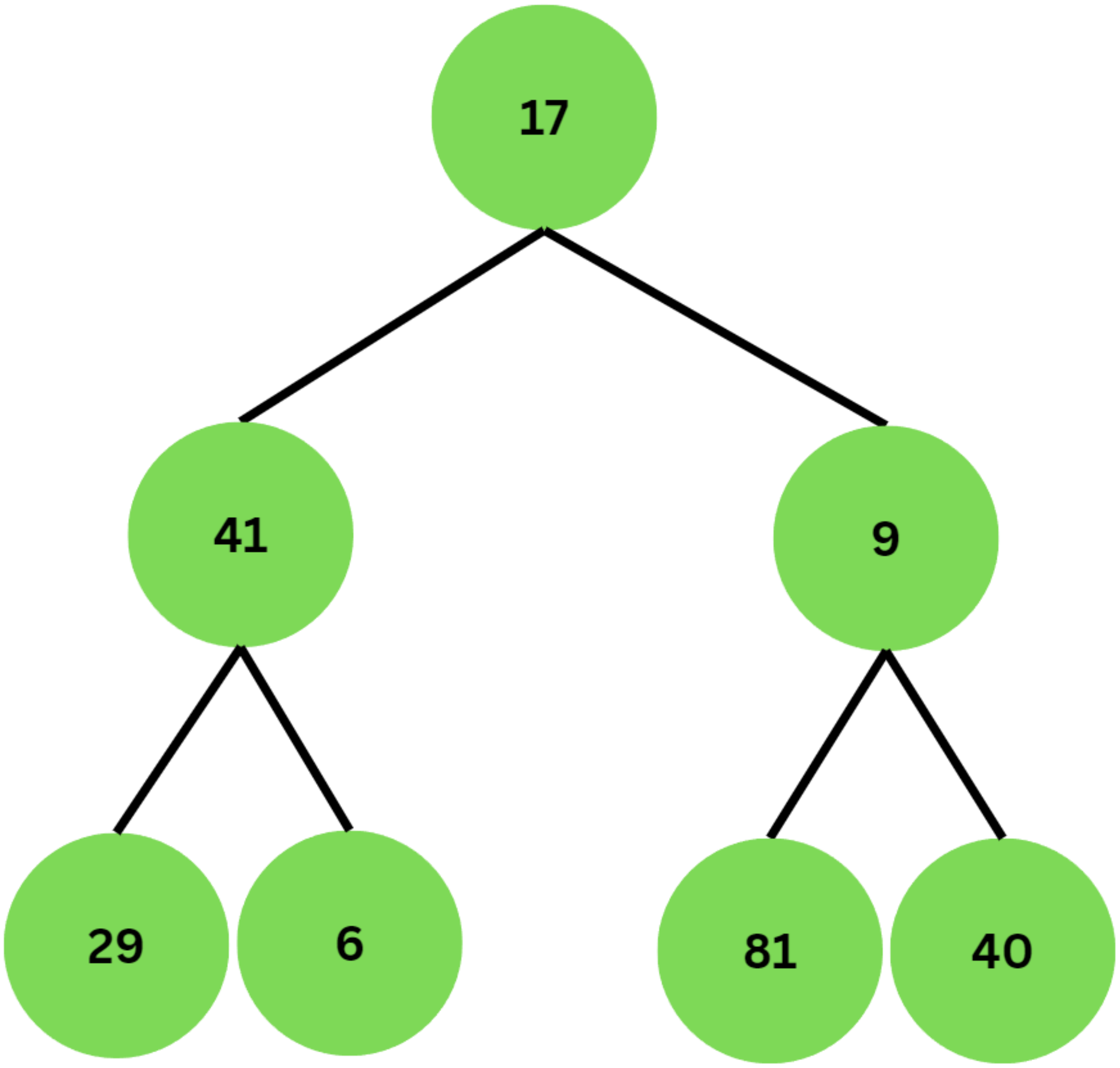
Binary Tree Traversal [Background Reading]

Types of Traversals

Depending on how you traverse through a binary tree, you will get different output. There are three main types of binary tree traversals:

- pre-order: In a pre-order traversal, the current node is visited first, followed by the left and right children.
- in-order: In an in-order traversal, the left child is visited first, followed by the current node and the right child.
- post-order: In a post-order traversal, the left and right children are visited first, followed by the current node.

For example, consider the following binary tree:



Here is the output depending on traversal:

- pre-order: 17 41 29 6 9 81 40
- in-order: 29 41 6 17 81 9 40
- post-order: 29 6 41 81 40 9 17

We can express these types of traversal with the following code:

```
private void printPreorder(IntTreeNode root) {
    if (root != null) {
       System.out.print(root.data + " ");
       printPreorder(root.left);
       printPreorder(root.right);
    }
}
```

```
private void printInorder(IntTreeNode root) {
    if (root != null) {
        printInorder(root.left);
        System.out.print(root.data + " ");
        printInorder(root.right);
    }
}

private void printPostorder(IntTreeNode root) {
    if (root != null) {
       printPostorder(root.left);
       printPostorder(root.right);
       System.out.print(root.data + " ");
    }
}
```

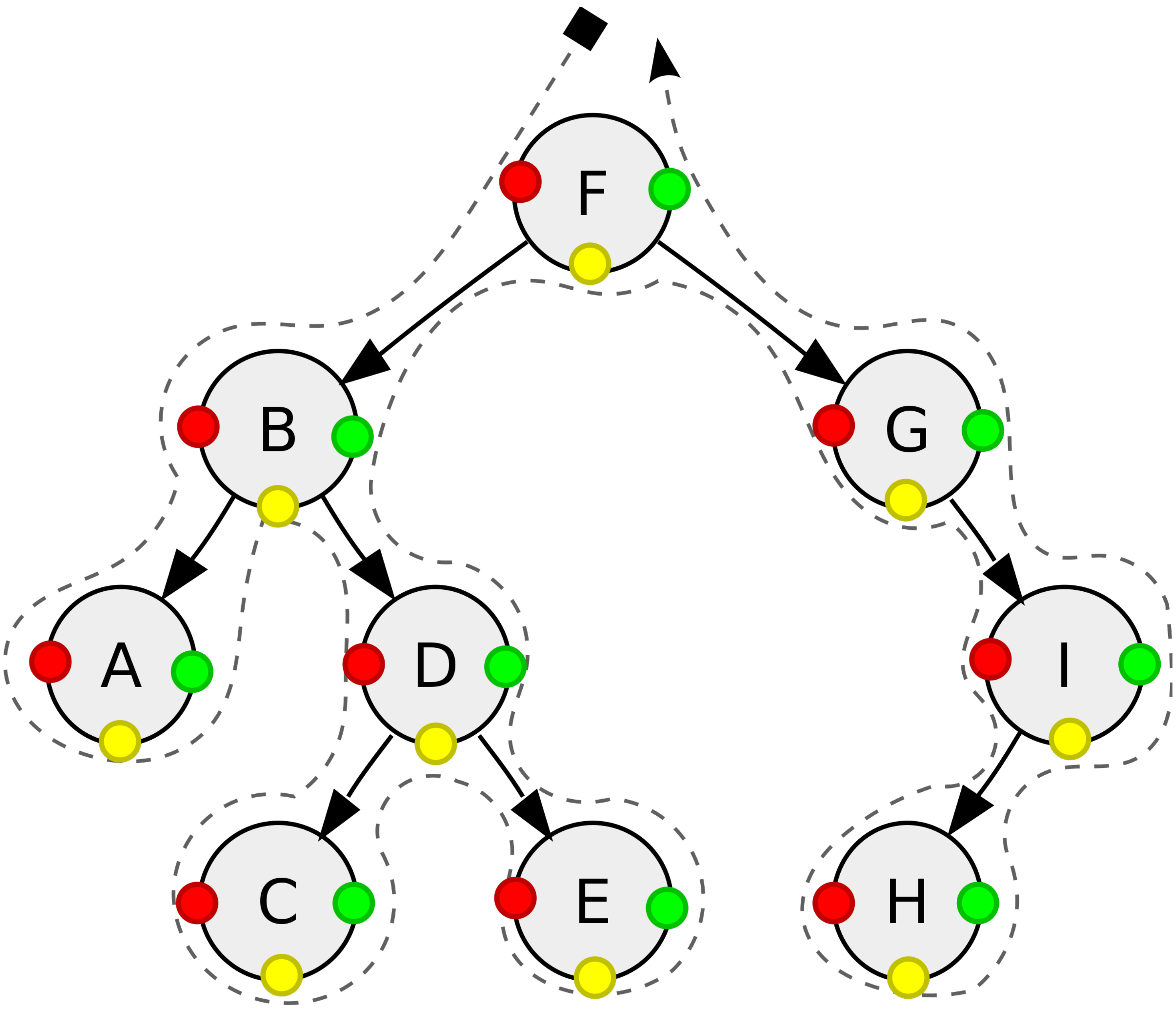
□ Sailboat Approach

There is a binary tree traversal trick which can help you quickly determine the pre-order, in-order, and post-order output called the sailboat approach. Imagine that each node is an island and that your cursor is the sailboat.

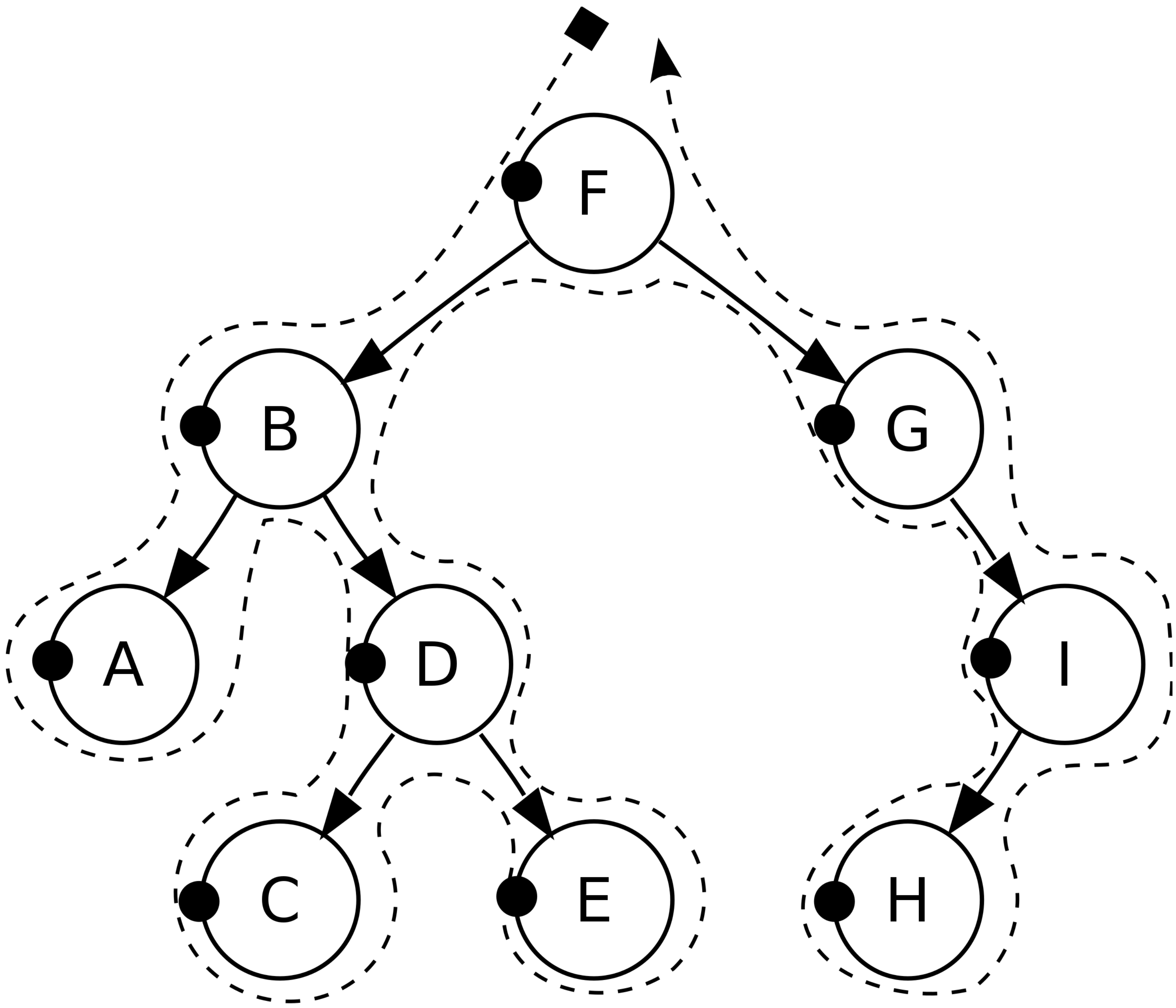
The main idea is to start at the root and move left-downwards, tracing a path around the tree (the islands). As you pass each node on the proper side, you will process it. To find out a certain traversal, you can mark a node on a certain side:

- pre-order: left side
- in-order: bottom
- post-order: right side

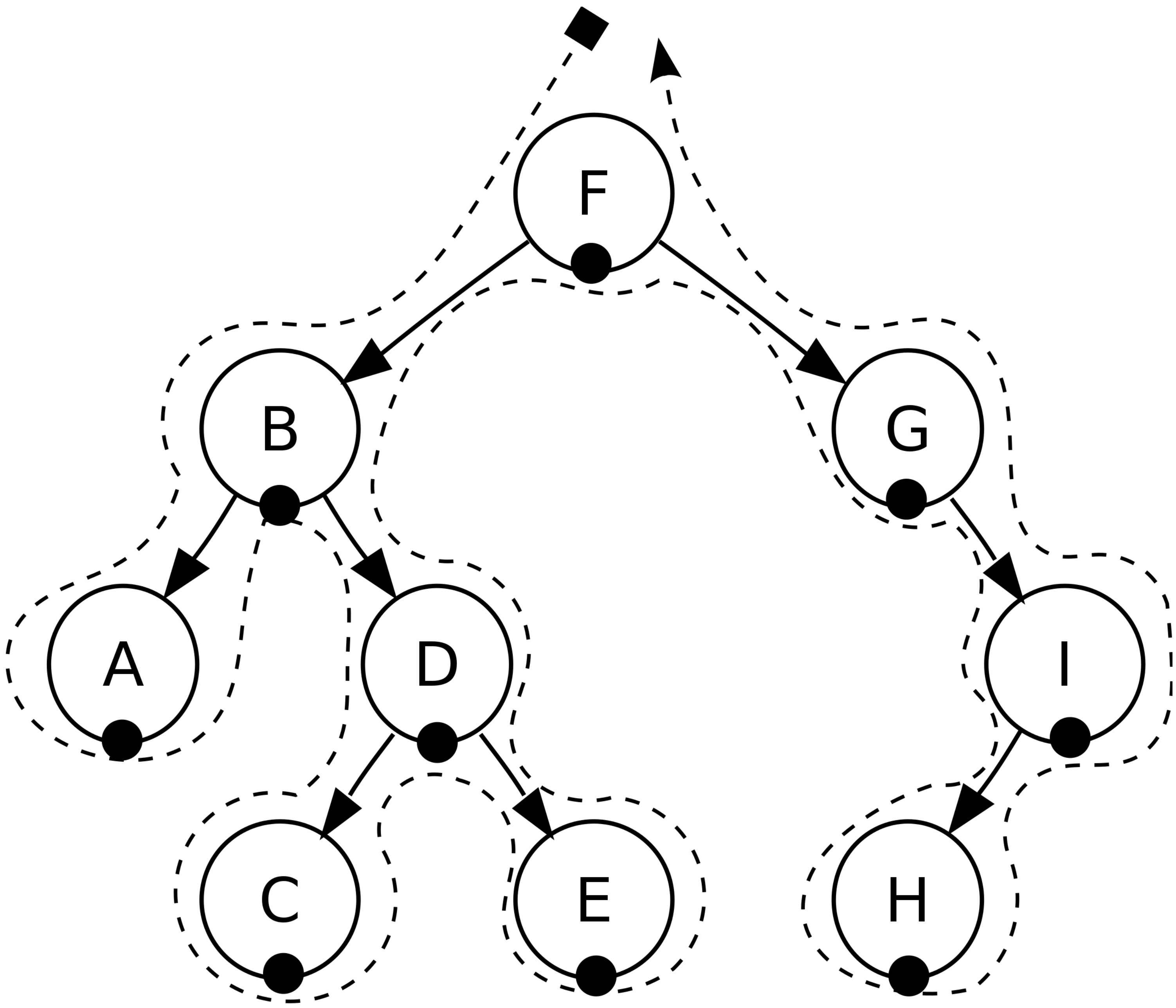
Suppose we had the following binary tree:



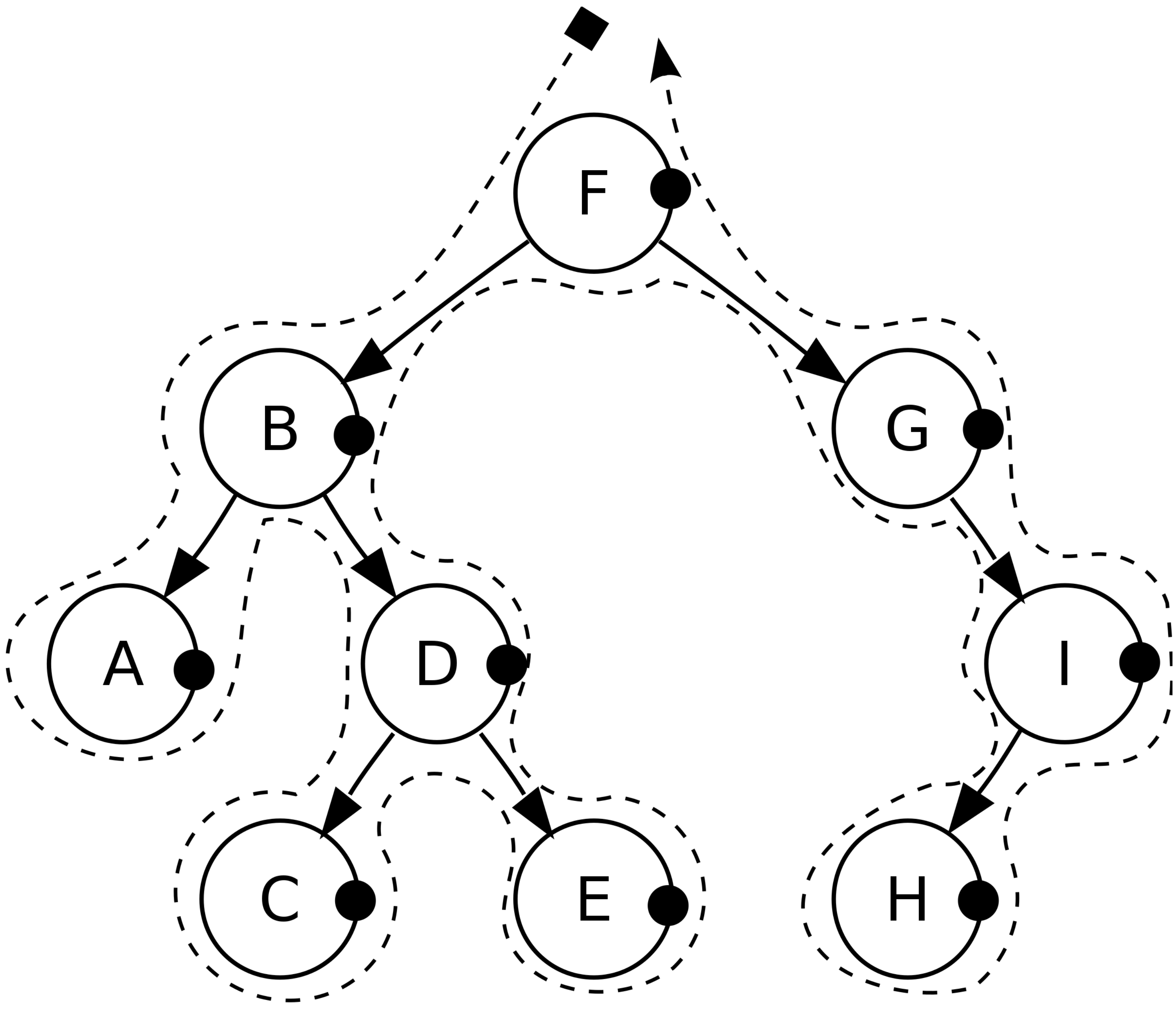
A pre-order traversal would give us: F B A D C E G I H. Here is what a pre-order traversal would look like using the sailboat approach:



A in-order traversal would give us: A B C D E F G H I. Here is what a in-order traversal would look like using the sailboat approach:



A post-order traversal would give us: A C E D B H I G F. Here is what a post-order traversal would look like using the sailboat approach:



🚩 Main Points

- Depending on how you traverse through a binary tree, you will get different output. There are three main types of traversals and the **sailboat approach** can help us determine their output.
    - *Pre-order traversal*
        - The current node is visited first, followed by the left and right children.
        - 🚩 -> Trace a path around the tree (the islands) and mark each node on the left side.
    - *In-order traversal*
        - The left child is visited first, followed by the current node and the right child.
        - 🚩 -> Trace a path around the tree (the islands) and mark each node on the bottom.
    - *Post-order traversal*
        - The left and right children are visited first, followed by the current node.

        - ▯ -> Trace a path around the tree (the islands) and mark each node on the right side.

Binary Tree Traversal [Practice Problem]

Consider the following tree.

```
          +---+
          | 3 |
          +---+
         /     \
     +---+     +---+
     | 5 |     | 2 |
     +---+     +---+
    /         /     \
+---+     +---+     +---+
| 1 |     | 4 |     | 6 |
+---+     +---+     +---+
```

Question 1

Give the elements of the tree below in the order they would be printed by a **pre-order traversal**. Separate each number with a single space.

No response

Question 2

Give the elements of the tree below in the order they would be printed by a **in-order traversal**. Separate each number with a single space.

No response

Question 3

Give the elements of the tree below in the order they would be printed by a **post-order traversal**. Separate each number with a single space.

No response

size

Write a method `size` that returns the total number of nodes in the tree.

The following tree has a size of 9.

```
            +---+
            | 5 |
            +---+
           /    \
        +---+   +---+
        | 3 |   | 6 |
        +---+   +---+
       /    \       \
    +---+   +---+   +---+
    | 2 |   | 4 |   | 7 |
    +---+   +---+   +---+
   /               /    \
+---+           +---+   +---+
| 1 |           | 8 |   | 9 |
+---+           +---+   +---+
```