## Lost in a Forest of Trees

### Step 1

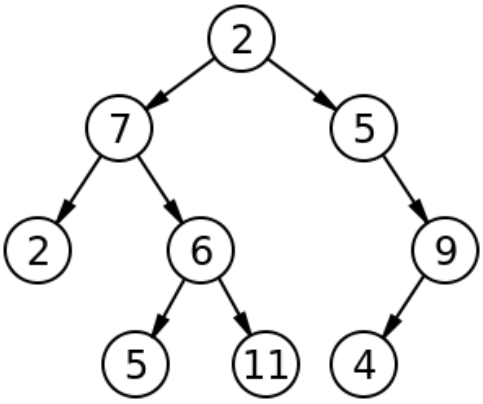
In the previous chapter, one of the introductory data structures we covered was the **Linked List**, which, as you recall, was a set of **nodes** that were strung together via links (**edges**). We had direct access to the "head" node (and the "tail" node as well, if we so chose), but in order to access nodes in the middle of the list, we had to start at the "head" (or "tail") node and traverse edges until we reached the node of interest.

The general structure of "a set of **nodes** and **edges**" is called a **Graph**, with a **Linked List** being an extremely simple example of a **Graph**. Specifically, we can think of a **Singly Linked List** as a **Graph** in which every node (except the "tail" node) has a single "forward" edge, and we can think of a **Doubly Linked List** as a **Graph** in which every node (except the "head" and "tail" nodes) has exactly two edges: one "forward" edge and one "back" edge.

The **Linked List** was a very simple **Graph** structure, so in this chapter, we'll introduce a slightly more complex **Graph** structure: the **Tree** structure.

### Step 2

Formally, a **graph** is, by definition, a collection of **nodes** (or **vertices**) and **edges** connecting these nodes. A **tree** is defined as a graph without any **undirected cycles** (i.e., cycles that would come about if we replaced directed edges with undirected edges) nor any **unconnected parts**. Below is an example of a valid tree:



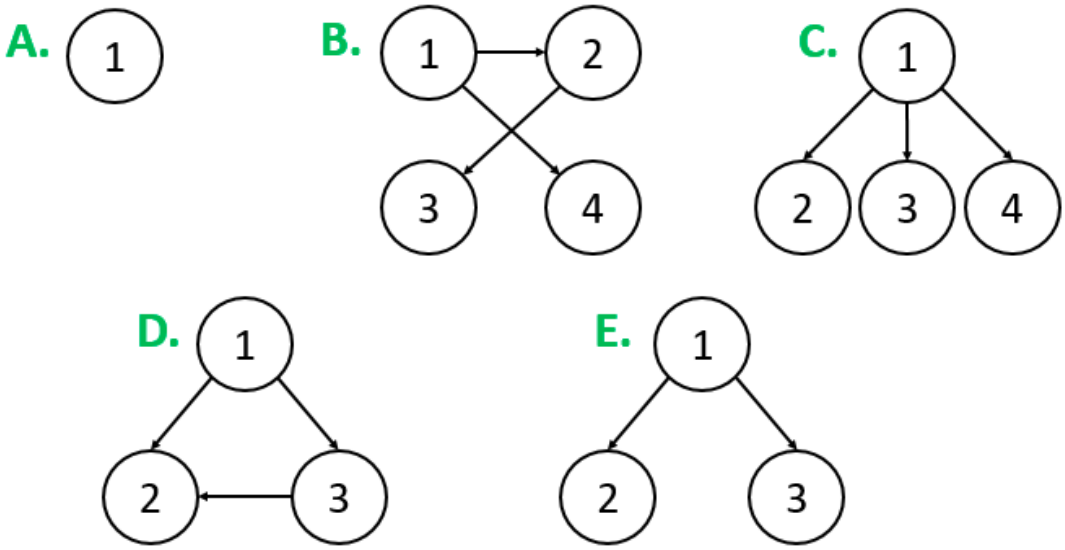

Note that, even though we drew this tree with directed edges, it is perfectly valid for a tree to have undirected edges.

Even though the tree above looks like what we would expect of a tree, there are some special "weird" cases that one might not expect are valid trees:

- a tree with no nodes is a valid tree called the "null" (or "empty") tree.
- a tree with a single node (and no edges) is also a valid tree.

### Step 3

**EXERCISE BREAK:** Which of the following are valid trees? (Select all that apply)



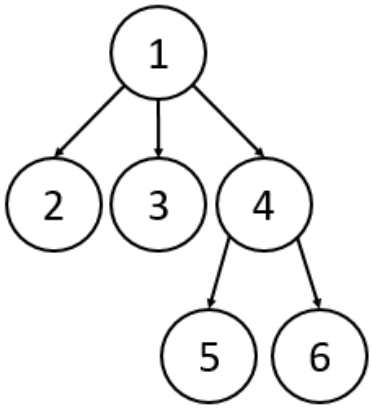

**To solve this problem please visit https://stepik.org/lesson/28726/step/3**

## Step 4

There are two classes of trees: **rooted** and **unrooted**.

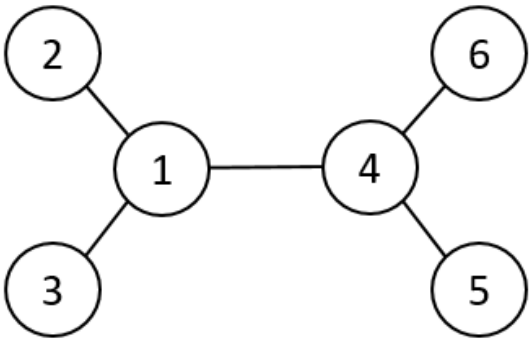
In a **rooted** tree, a given node can have a single **parent** node above it and can have any number of **children** nodes below it. Just like with a family tree, all nodes that appear along the path going upward from a given node are considered that node's **ancestors**, and all nodes that appear along any path going downward from a given node are considered that node's **descendants**. There is a single node at the top of the tree that does not have a *parent*, which we call the **root**, and there can be any number of nodes at the bottom of the tree that have no *children*, which we call **leaves**. All nodes that have *children* are called **internal nodes**.

Below is an example of a *rooted* tree, where the *root* is 1, the *leaves* are {2, 3, 5, 6}, and the *internal nodes* are {1, 4}:




In an **unrooted** tree, there is no notion of *parents* or *children*. Instead, a given node has **neighbors**. Any nodes with just a single *neighbor* is considered a **leaf** and any node with more than one neighbor is considered an **internal node**.

Below is an example of an *unrooted* tree, where the *leaves* are {2, 3, 5, 6} and the *internal nodes* are {1, 4}:

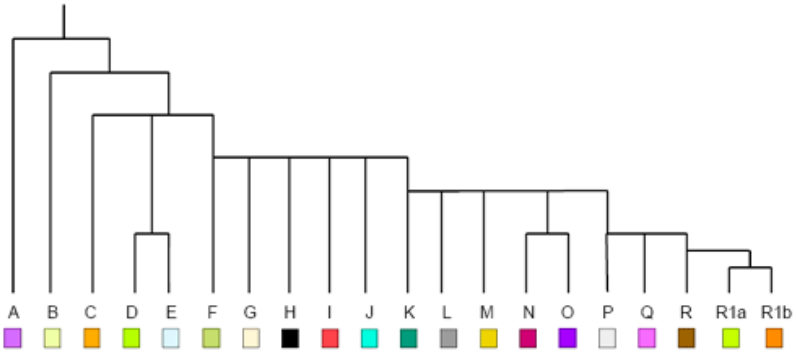



**STOP AND THINK:** If I were to make the directed edges of the first tree into undirected edges, would it become equivalent to the second tree?

## Step 5

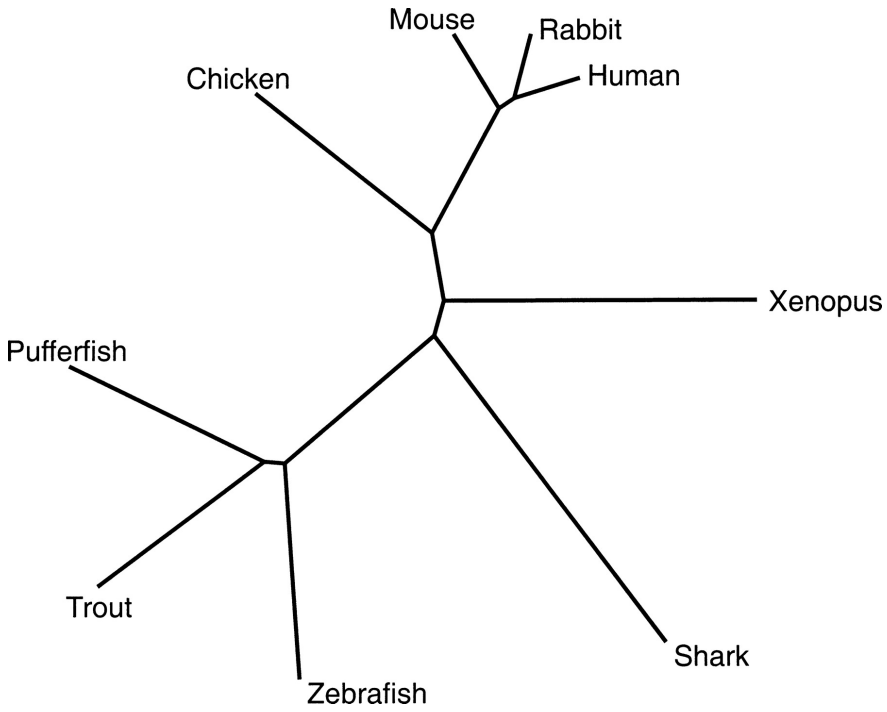
**Rooted trees** have an implied hierarchical structure: an arbitrary node may have *children* and *descendants* below it, and it may have a *parent* and *ancestors* above it. In some contexts, we might want to interpret this hierarchy as "top-down" (i.e., we care about relationships from root to leaves) or as "bottom-up" (i.e., we care about relationships from leaves to root), but either way, the underlying tree structure is the same, despite how we choose to interpret the hierarchy it can represent.

For example, in Bioinformatics, we can study a paternal lineage of humans by creating an evolutionary tree from the Y chromosomes of a large set of individuals (because the Y chromosome is a single DNA sequence that is passed down from father to son). In this case, the tree represents time in the "top-down" direction: traversing the tree from *root* to *leaves* takes you forward in time. As a result, the *root* node is the most recent common ancestor of the individuals we chose to sequence. Below is an example of such a tree:




**Unrooted trees** do not necessarily have an implied hierarchical structure. In some contexts, we might want to interpret an "inside-out" hierarchy (i.e., we care about relationships from internal nodes to leaves) or an "outside-in" hierarchy (i.e., we care about relationships from leaves to internal nodes). In unrooted trees, instead of thinking of nodes as "parents" or "children," it is common to simply think of nodes as "neighbors". Also, even if we don't impose any sense of directionality onto the tree, distances between nodes can provide some information of "closeness" or "relatedness" between them.

For example, in Bioinformatics, if we want to construct an evolutionary tree from different organisms, if we don't know what the true ancestor was, we have no way of picking a node to be the root. However, the unrooted tree still provides us relational information: leaves closer together on the tree are more similar biologically, and leaves farther apart on the tree are more different biologically. Below is an example of such a tree:




## Step 6

For a graph *T* to be considered a tree, it must follow the all of the following constraints (which can be proven to be equivalent constraints):

- *T* is connected and has no undirected cycles (i.e., if we made *T*'s directed edges undirected, there would be no cycles)
- *T* is acyclic, and a simple cycle is formed if any edge is added to *T*
- *T* is connected, but is not connected if any single edge is removed from *T*
- There exists a unique simple path connecting any two vertices in *T*

If *T* has *n* vertices (where *n* is a finite number), then the above statements are equivalent to the following two conditions:

- *T* is connected and has $n-1$ edges
- *T* has no simple cycles and has $n-1$ edges

However, we run into a bit of a problem with the 0-node graph, which we previously called the "empty" tree. On the one hand, as a graph, it violates some of the constraints listed above and thus should not be considered a tree. On the other hand, we explicitly called it a valid tree data structure previously because it represents a tree data structure containing 0 elements, which is valid. What is the correct verdict? For the purposes of this text, since we are focusing on data structures and not on graph algorithms specifically, let's explicitly handle this edge case and just say that a 0-node graph is a valid tree, despite the fact that it fails some of the above conditions.

**STOP and Think:** Which of the above conditions are violated by the empty tree?

## Step 7

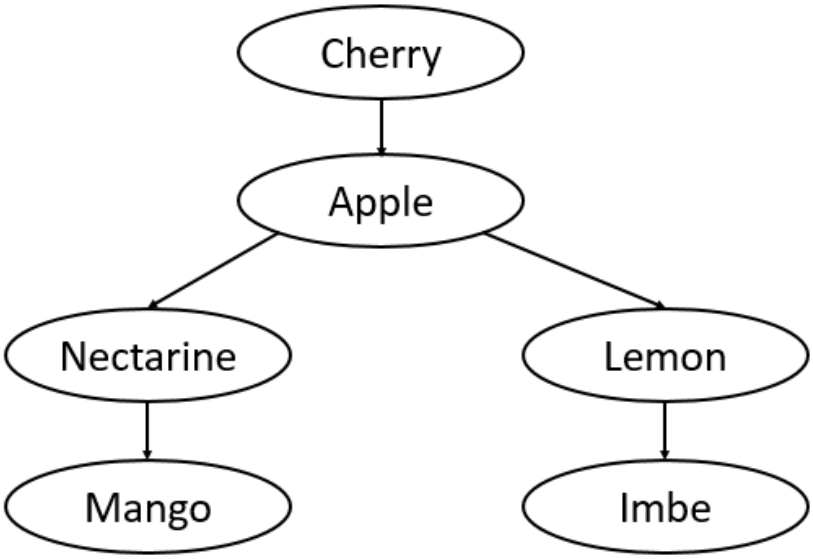
**CODE CHALLENGE: Recursively Visiting the Nodes of a Tree**

We have defined the following C++ class for you:

```
class Node {
    public:
        string label;
        vector<Node*> children;
};
```

Write a function `recursivePrint(Node* node)` that recursively prints the labels of all nodes in the subtree rooted at `node` (in any order). For example, if we had a tree with some root node `root`, then calling `recursivePrint(root)` would print the labels of all nodes in the tree (in some arbitrary order). Print a single label per line.

The tree represented by the Sample Input has been reproduced below for clarity:




**Sample Input:**

```
Cherry -> Apple
Apple -> Nectarine
Apple -> Lemon
Nectarine -> Mango
Lemon -> Imbe
```

**Sample Output:**

```
Lemon
Cherry
Apple
Imbe
Mango
Nectarine
```

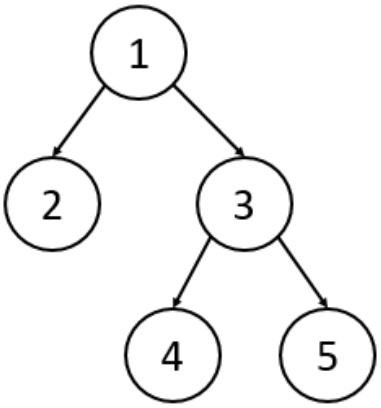
**To solve this problem please visit https://stepik.org/lesson/28726/step/7**

## Step 8

In this text, we will pay special attention to **rooted binary trees**, which are rooted trees with the following two restrictions:

- All nodes except the root have a parent
- All nodes have either 0, 1, or 2 children

Many of the example trees used previously are binary trees, but an example binary tree has been reproduced below:

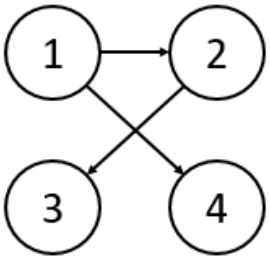


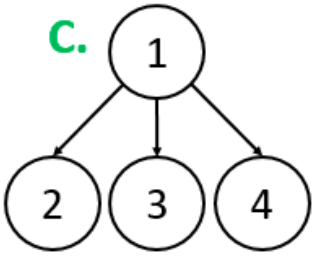

Note that *any* graph that follows the above restrictions is considered a valid rooted binary tree. Again, for our purposes, we will consider the empty tree to be a valid rooted binary tree. Also, we specify "rooted" binary tree here because there do exist "unrooted" binary trees, but they are out of the scope of this text, so if we use the term "binary tree" at any point later in this text without specifying "rooted" or "unrooted," we mean "rooted".

## Step 9

**EXERCISE BREAK:** Which of the following are valid rooted binary trees? (Select all that apply)

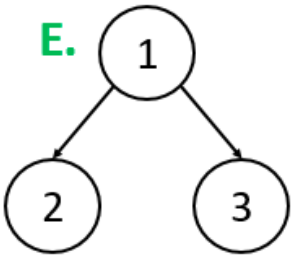
**A.** (empty)

**B.**



**C.**



**D.**



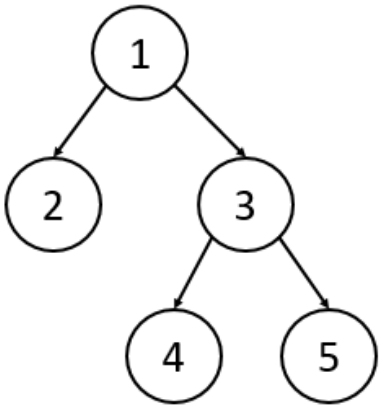
**E.**



**To solve this problem please visit https://stepik.org/lesson/28726/step/9**

## Step 10

With rooted binary trees (and rooted trees in general), we typically only maintain a pointer to the root because all other nodes in the tree can be accessed via some traversal starting at the root. It can be useful to keep pointers to the leaves of the tree as well, but these pointers can be harder to maintain because the leaves of a rooted tree change rapidly as we insert elements into the tree.

Because we typically only keep track of the root node, to traverse all of the nodes in a rooted binary tree, there are four traversal algorithms: **pre-order**, **in-order**, **post-order**, and **level-order**. In all four, the verb "visit" simply denotes whatever action we perform when we are at a given node *u* (whether it be printing *u*'s label, or modifying *u* in some way, or incrementing some global count, etc.).




In a **pre-order** traversal, we first visit the current node, then we recurse on the left child (if one exists), and then we recurse on the right child (if one exists). Put simply, **VLR** (**V**isit-**L**eft-**R**ight). In the example above, a **pre-order** traversal starting at the root would visit the nodes in the following order: 1 2 3 4 5

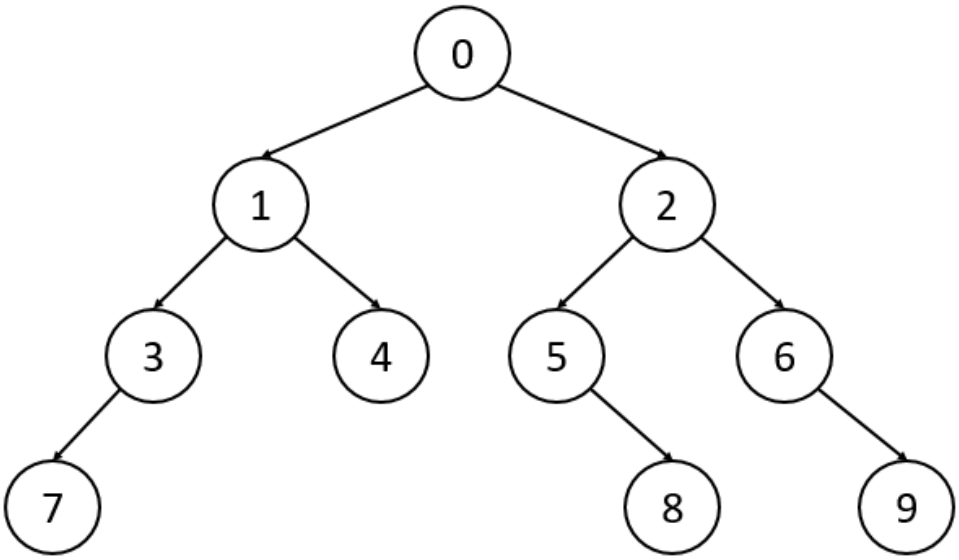
In an **in-order** traversal, we first recurse on the left child (if one exists), then we visit the current node, and then we recurse on the right child (if one exists). Put simply, **LVR** (**L**eft-**V**isit-**R**ight). In the example above, an **in-order** traversal starting at the root would visit the nodes in the following order: 2 1 4 3 5

In a **post-order** traversal, we first recurse on the left child (if one exists), then we recurse on the right child (if one exists), and then we visit the current node. In the example above, a **post-order** traversal starting at the root would visit the nodes in the following order: 2 4 5 3 1

In a **level-order** traversal, we visit nodes level-by-level (where the root is the "first level", its children are the "second level", etc.), and on a given level, we visit nodes left-to-right. In the example above, a **level-order** traversal starting at the root would visit the nodes in the following order: 1 2 3 4 5

## Step 11

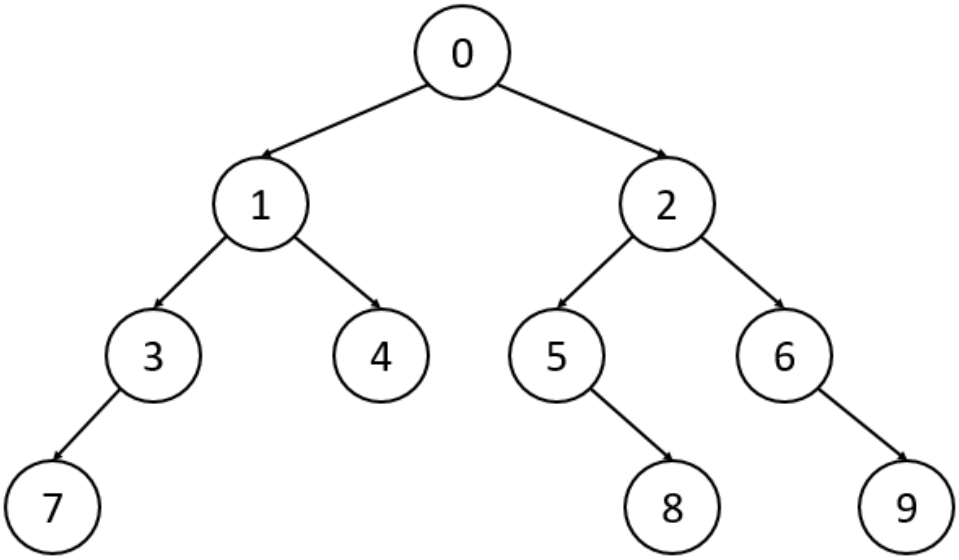
**EXERCISE BREAK:** In what order will the nodes of the following tree be visited in a **post-order** traversal? Separate your output using single space characters (e.g. 1 2 3 4 5)




**To solve this problem please visit https://stepik.org/lesson/28726/step/11**

## Step 12

**EXERCISE BREAK:** In what order will the nodes of the following tree be visited in a **level-order** traversal? Separate your output using single space characters (e.g. 1 2 3 4 5)




**To solve this problem please visit https://stepik.org/lesson/28726/step/12**

## Step 13

**EXERCISE BREAK:** What is the worst-case time complexity of performing a **pre-order**, **in-order**, **post-order**, or **level-order** traversal on a rooted binary tree with *n* nodes? (All four traversals have the same worst-case time complexity)

**To solve this problem please visit https://stepik.org/lesson/28726/step/13**

## Step 14

Now that you have learned about trees in general, with a focus on rooted binary trees, you are ready to learn about the various other specific tree structures covered in this text. The bulk of these trees will be rooted binary trees, so don't forget the properties and algorithms you learned in the past few steps!