Tree Depth-first Search: Introduction

Let's understand the Tree Depth-first Search pattern, its real-world applications, and some problems we can solve with it.

We'll cover the following

Overview

A tree is a graph that contains the following properties:

- It is undirected.
- It is acyclic (contains no cycles).
- It has a single connected component.

We know that a tree is a data structure that can have values of any data type and is made up of nodes, connected by edges. However, what sets a tree apart from other data structures, such as an array or linked list, is that multiple ways exist to explore it. **Tree depth-first search** is a method to reduce the use of nested loops in our tree-related problems. Depth-first search in a tree is usually implemented recursively. However, it can also be implemented iteratively with the help of a stack. The fundamental characteristic of this pattern is that it travels as far as possible along each tree branch before exploring the others.

There are three main methods to solving a problem with the depth-first search pattern—Preorder, Inorder, and Postorder. We'll take a look at each of them briefly and talk about how they are used—and more importantly, where they are used:

- **Preorder traversal:** The preorder traverses the tree in the following three steps:
  - Visit the root node
  - Recursively perform preorder traversal on the left child
  - Recursively perform pre-order traversal on the right child

This traversal is the most commonly used variation of the depth-first search and probably the one you'll use most of the time. It means that we start by visiting the root of the tree itself and then traverse our tree by first exploring all the branches on the left side until we reach their deepest nodes, and then repeating the process for the nodes on the right side of our root.

> The meaning of *visit* depends on the application. If we are calculating the size in bytes of the directory structure on a disk, for example, this may involve accumulating the size of the files in a particular directory.

- **Inorder traversal:** The inorder tree traversal explores the tree in the following three steps:
  - Recursively perform inorder traversal on the left child
  - Visit the node
  - Recursively perform inorder traversal on the right child

Inorder traversal means that we move from the leftmost element to the rightmost element in our tree. In other words, we would start traversing our tree from the left first, before exploring the right side of the tree. Thi

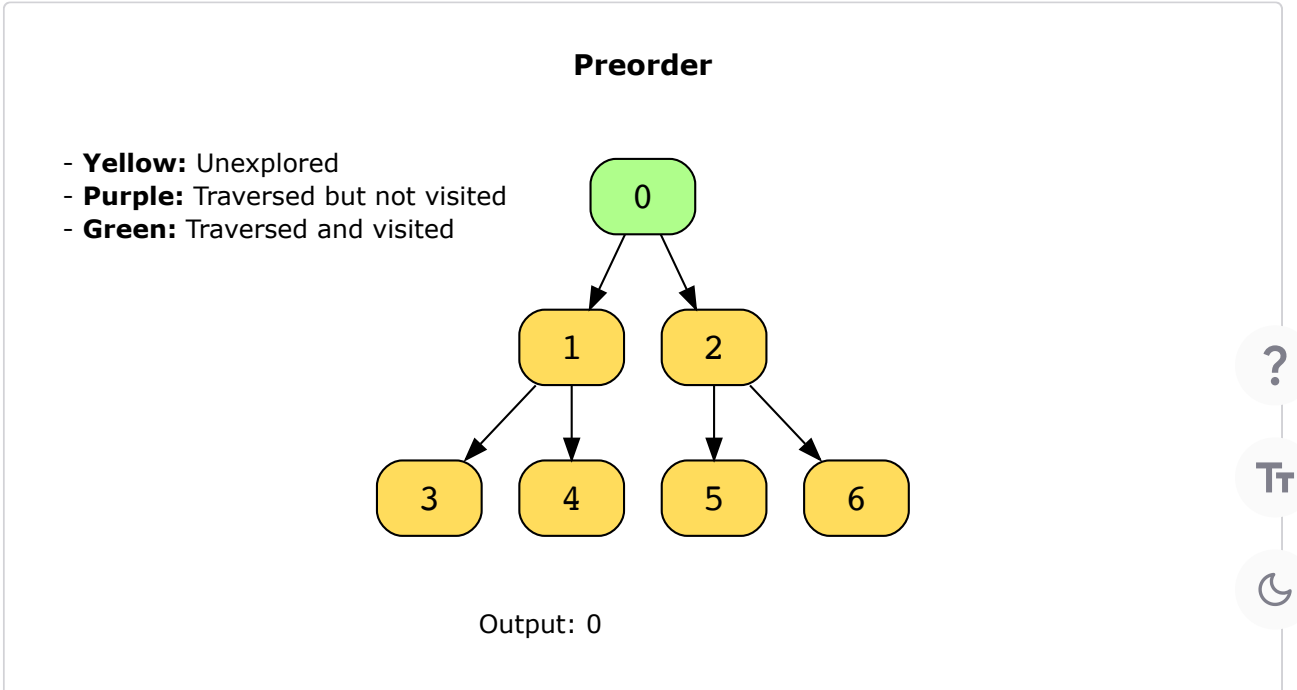
variation of depth-first search is usually used when we have to find elements that are smaller than the root in a binary search tree.

- **Postorder traversal:** The postorder traversal explores the tree in the following three steps:
    - Recursively perform postorder traversal on the left child
    - Recursively perform postorder traversal on the right child
    - Visit the node

This traversal is perhaps the rarest of the three when it comes to solving problems. It works by going to the deepest node on the left side of the tree and storing that node. If a node does not have a left child, it goes to the right child and visits it. Finally, it visits the parent node.

Imagine deleting a tree. If we use preorder traversal and delete the root node, first, we would lose the pointers to the left and right subtrees and wouldn't be able to delete those. That's a memory leak. If we use inorder traversal and delete the left child followed by the root, the right subtree couldn't be deleted. If we use the postorder traversal and delete the root node after deleting both the subtrees, then we get the desired result.

We can take a look at how each of these methods works in the example below:



We have a clear understanding of how depth-first search works, so now we can talk about the more critical part—why we need it. A naive approach to exploring the tree would be to revisit already traversed nodes. More specifically, we start at the root node and traverse a particular branch all the way to the leaf node. Then, we start at the root node again and explore another branch. In this way, we will revisit many nodes over and over. This would take our time complexity to $O(n^2)$ in the worst case. However, if we use the tree depth-first search pattern, we are guaranteed to reach a solution in $O(n)$ time complexity unless there are some significant constraints!

A frequent question asked at this point by learners is what they should choose between breadth-first search (BFS) and depth-first search (DFS). DFS starts at the root and traverses nodes as far as possible until we reach a leaf node that contains no left or right children. On the contrary, BFS traverses nodes on the same level before moving to the next level. Therefore, we would prefer DFS if the destination node were close to a leaf. Meanwhile, we would favor BFS if our goal is to search for a node that is more likely to be closer to the root itself.

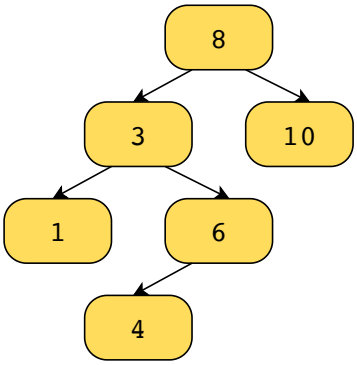
Examples

The following examples illustrate some problems that can be solved with this approach:

Determine if a tree is a valid binary search tree

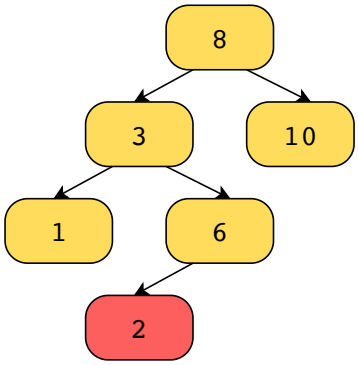
We perform an inorder traversal of the tree. If the value of the current node is less than the value of the previous node traversed, the tree is not a valid BST, so we return FALSE. Otherwise, we keep on traversing the tree and return TRUE if the traversal of the entire tree is successful.

Valid BST



Inorder traversal: 1, 3, 4, 6, 8, 10

Invalid BST



Inorder traversal: 1, 3, 2, 6, 8, 10

1 of 2

Does my problem match this pattern?

- Yes, if either of these conditions is fulfilled:
  - We have reason to believe that the solution is near the leaves of the tree.
  - Components of the solution are listed along paths from the root to the leaves and finding the optimal solution requires traversal along these paths. The classic example of this is finding the height of the given tree.
- No, if either of these conditions is fulfilled:
  - The input data is not in the form of a tree, or the cost of transforming it into a tree is too high, given the specified efficiency requirements.

Real-world problems

Many problems in the real world use the tree depth-first search pattern. Let's look at some examples.

Find products in a price range: Convert the prices of all products into a binary search tree and perform a variation of preorder traversal on the tree. We start at the root and check if its value lies in the range. If it does, we add it to the output array. We then check if the value is greater than or equal to the lower bound. If it is, we will call the preorder traversal on the left child of the node. If the node's value is less than or equal to the upper bound, we'll also call the preorder traversal on the right child of the node. We repeat this process until the tree has been traversed and then return the output array.

Finding routes: Used in maps for finding a route between a start and destination point.

Solving mazes: Used for finding the path out of a maze. We select a path and follow it until we hit a dead end. If we do, we backtrack and take an alternative path from the past junction and try this new path. We repeat this process until the destination is reached.

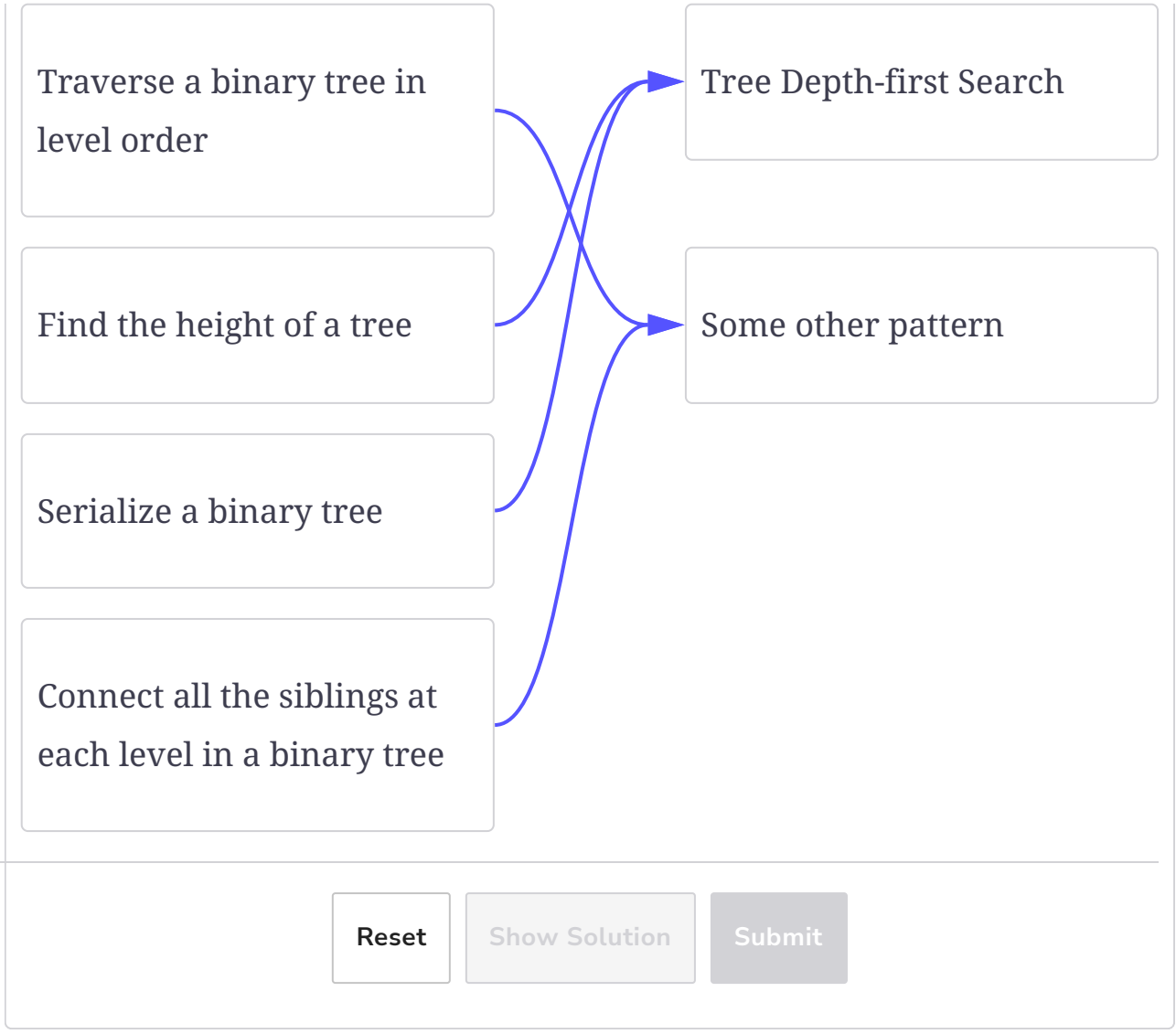
Strategy time!

Match the problems that can be solved using the tree depth-first search pattern.

> **Note:** Select a problem in the left-hand column by clicking it, and then click one of the two options in the right-hand column.

Match The Answer

Select an option from the left-hand side


Traverse a binary tree in level order
Find the height of a tree
Serialize a binary tree
Connect all the siblings at each level in a binary tree
Tree Depth-first Search
Some other pattern
Reset
Show Solution
Submit

Back
Next