Binary trees - DFS

Here we'll talk about how to **traverse** binary trees. Tree traversal is how we access the elements of a tree, and thus is mandatory for solving tree problems.

Recall that we traversed a linked list using code similar to:

Python3

```
def get_sum(head):
    ans = 0
    while head:
        ans += head.val
        head = head.next

    return ans
```

JavaScript

```
let getSum = head => {
    let ans = 0;
    while (head) {
        ans += head.val;
        head = head.next;
    }

    return ans;
}
```

Java

```
int getSum(ListNode head) {
    int ans = 0;
    while (head != null) {
```

```
            ans += head.val;
            head = head.next;
        }

        return ans;
    }
```

C++

```
int getSum(ListNode* head) {
    int ans = 0;
    while (head != nullptr) {
        ans += head->val;
        head = head->next;
    }

    return ans;
}
```

This code starts at the `head` and visits each node to find the sum of all the values in the linked list.

When traversing linked lists, we usually do it iteratively. With binary trees, we usually do it recursively.

There are two main types of tree traversals. The first is called depth-first search (DFS). for binary trees specifically, there are 3 ways to perform DFS - preorder, inorder, and postorder (the type you choose rarely matters for solving problems). The other main type of traversal is called breadth-first search (BFS).

Depth-first search (DFS)

Recall that the depth of a node is its distance from the root.

In a DFS, we prioritize depth by traversing as far down the tree (as deep) as possible in one direction (until reaching a leaf node) before considering the other direction. For example, let's say we choose left as our priority direction. We move exclusively with `node.left` until the left subtree has been fully explored. Then, we explore the right subtree.

Once a tree fully explores a branch, it "backtracks" until it finds another unexplored branch.

Because we need to backtrack up the tree after reaching the end of a branch, DFS is typically implemented using recursion, although it is also sometimes done iteratively using a stack. Here is an example of recursive DFS to visit every node:

Python3

```
def dfs(node):
    if node == None:
        return

    dfs(node.left)
    dfs(node.right)
    return
```

JavaScript

```
let dfs = node => {
    if (!node) {
        return;
    }

    dfs(node.left);
    dfs(node.right);
    return;
}
```

Java

```
public void dfs(Node node) {
    if (node == null) {
        return;
    }

    dfs(node.left);
    dfs(node.right);
```

```
    return;
}
```

C++

```
void dfs(TreeNode* node) {
    if (node == nullptr) {
        return;
    }

    dfs(node->left);
    dfs(node->right);
    return;
}
```

The structure for performing a DFS is very similar across problems:

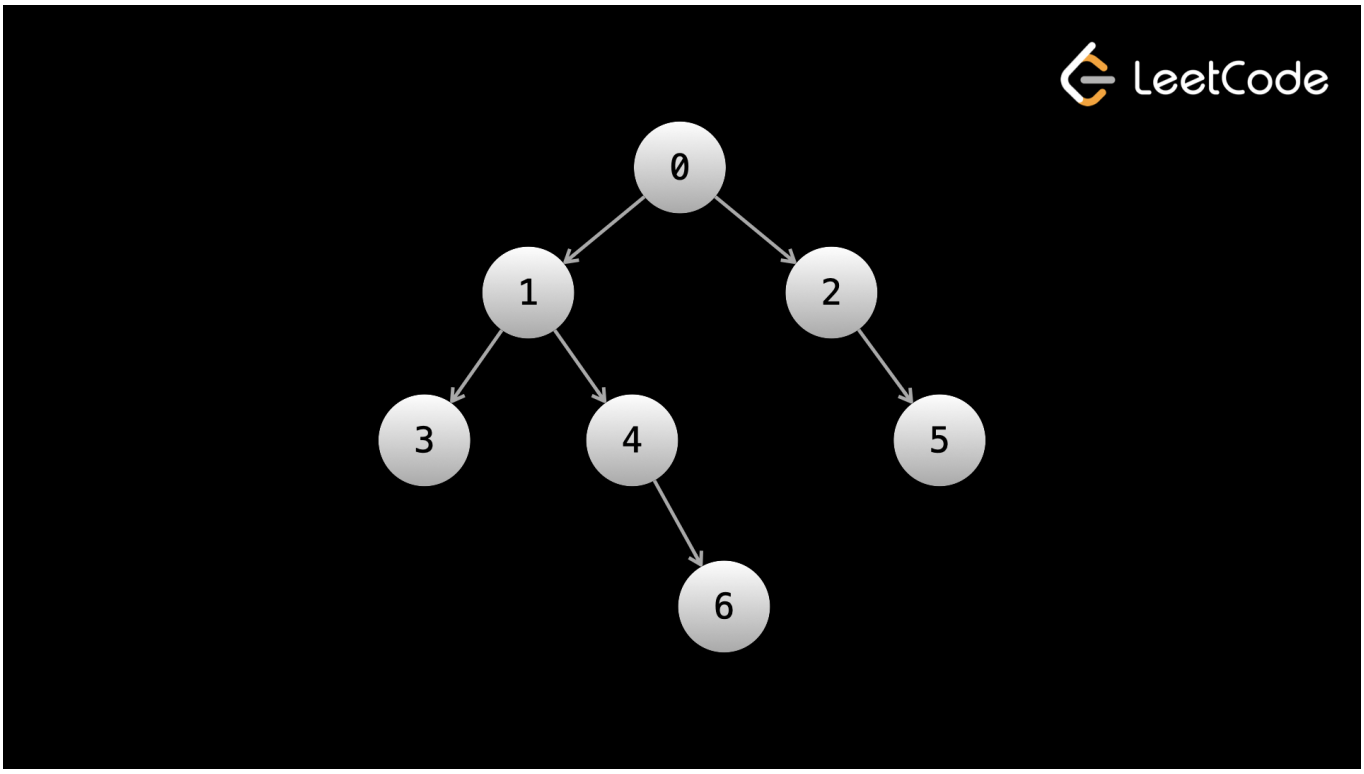
1. Handle the base case(s). Usually, an empty tree (`node = null`) is a base case.
2. Do some logic for the current node.
3. Recursively call on the current node's children
4. Return the answer

Steps 2 and 3 may happen in different orders.

The most important thing to understand when it comes to solving binary tree problems is that **each function call solves and returns the answer to the original problem as if the subtree rooted at the current node was the input.** The logic that will be done at each call (step 2) will depend on the problem.

Each of the three types of DFS differs only in the order in which they execute 2/3.

Consider the following binary tree:



Preorder traversal

In preorder traversal, logic is done on the current node before moving to the children. Let's say that we wanted to just print the value of each node in the tree to the console. In that case, we would print the current node's value, then recursively call the left child, then recursively call the right child.

Python3

```
def preorder_dfs(node):
    if not node:
        return

    print(node.val)
    preorder_dfs(node.left)
    preorder_dfs(node.right)
    return
```

JavaScript

```
let preorderDfs = node => {
    if (!node) {
        return;
    }

    console.log(node.val);
    preorderDfs(node.left);
    preorderDfs(node.right);
    return;
}
```

Java

```
public void preorderDfs(Node node) {
    if (node == null) {
        return;
    }

    System.out.println(node.val);
    preorderDfs(node.left);
    preorderDfs(node.right);
    return;
}
```

C++

```
void preorderDfs(TreeNode* node) {
    if (node == nullptr) {
        return;
    }

    cout << node->val << endl;
    preorderDfs(node->left);
    preorderDfs(node->right);
    return;
}
```

Running the above code on the example tree, we would see the nodes printed in this order: `0, 1, 3, 4, 6, 2, 5`.

Because the logic (printing) is done immediately at the start of each function call, preorder handles nodes in the same order that the function calls happen.

Inorder traversal

For inorder traversal, we first recursively call the left child, then perform logic (print in this case) on the current node, and then recursively call the right child. This means no logic will be done until we reach a node without a left child since calling on the left child takes priorty over performing logic.

Python3

```
def inorder_dfs(node):
    if not node:
        return

    inorder_dfs(node.left)
    print(node.val)
    inorder_dfs(node.right)
    return
```

JavaScript

```
let inorderDfs = node => {
    if (!node) {
        return;
    }

    inorderDfs(node.left);
    console.log(node.val);
    inorderDfs(node.right);
    return;
}
```

Java

```
public void inorderDfs(Node node) {
    if (node == null) {
        return;
    }

    inorderDfs(node.left);
    System.out.println(node.val);
    inorderDfs(node.right);
    return;
}
```

C++

```
void inorderDfs(TreeNode* node) {
    if (node == nullptr) {
        return;
    }

    inorderDfs(node->left);
    cout << node->val << endl;
    inorderDfs(node->right);
    return;
}
```

Running the above code on the example tree, we would see the nodes printed in the order: `3, 1, 4, 6, 0, 2, 5`.

For any given node, its value is not printed until all values in the left subtree are printed, and values in its right subtree are not printed until after that.

Postorder traversal

In postorder traversal, we recursively call on the children first and then perform logic on the current node. This means no logic will be done until we reach a leaf node since calling on the children takes priority over performing logic. In a postorder traversal, the root is the last node where logic is done.

Python3

```
def postorder_dfs(node):
    if not node:
        return

    postorder_dfs(node.left)
    postorder_dfs(node.right)
    print(node.val)
    return
```

JavaScript

```
let postorderDfs = node => {
    if (!node) {
        return;
    }

    postorderDfs(node.left);
    postorderDfs(node.right);
    console.log(node.val);
    return;
}
```

Java

```
public void postorderDfs(Node node) {
    if (node == null) {
        return;
    }

    postorderDfs(node.left);
    postorderDfs(node.right);
    System.out.println(node.val);
    return;
}
```

C++

```
void postorderDfs(TreeNode* node) {
    if (node == nullptr) {
        return;
    }

    postorderDfs(node->left);
    postorderDfs(node->right);
    cout << node->val << endl;
    return;
}
```

Running the above code on the example tree, we would see the nodes printed in this order: `3, 6, 4, 1, 5, 2, 0`.

Notice that for any given node, no values in its right subtree are printed until all values in the left subtree are printed, and its own value is not printed until after that.

The name of each traversal is describing when the current node's logic is performed:

Pre -> before children

In -> in the middle of children

Post -> after children

Example 1: 104 - Maximum Depth of Binary Tree

Given the `root` of a binary tree, find the length of the longest path from the root to a leaf.

Let's start with a recursive approach. When thinking about designing recursive functions, a good starting point is always the base case. What is the depth of an empty tree (zero nodes, root is `null`)? The depth is `0`.

But in this problem we include the root in the depth so the depth of a 1-node tree is 1.

The problem states that we are looking for a path from the root to a leaf, which means that at the current node, we can only consider either the left or right subtree, not both. If `maxDepth(node.left)` represents the maximum depth of the left subtree and

`maxDepth(node.right)` represents the maximum depth of the right subtree, then we should take the greater value and add `1` to it (because the current node contributes `1` to the depth).

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0

        left = self.maxDepth(root.left)
        right = self.maxDepth(root.right)
        return max(left, right) + 1
```

What about an iterative implementation?

In the solution above, we are doing a postorder traversal because the logic for the current node (basically just the return statement) happens after the calls. All three types of DFS can be implemented iteratively, but postorder and inorder are more complicated to implement than preorder (which is very easy). As mentioned earlier, for most problems, the type of DFS doesn't matter, so we'll take a look at a preorder DFS implemented iteratively.

To implement DFS iteratively, we need to use a stack. We don't have the return values to store the depths, so we will instead need to associate the current depth with each node on the stack. The method of pairing the values will depend on the language you are using. Of the most popular languages, it's easiest in Python as you can just store tuple literals in the stack. In other languages like Java, you need to use two stacks or create a pair object or some other method.

The code format for implementing DFS iteratively is also very similar across most problems.

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
```

```
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0

        stack = [(root, 1)]
        ans = 0

        while stack:
            node, depth = stack.pop()
            ans = max(ans, depth)
            if node.left:
                stack.append((node.left, depth + 1))
            if node.right:
                stack.append((node.right, depth + 1))

        return ans
```

The format for performing the traversal with the stack is something that can be easily re-used between problems. We use a stack and a while loop until the stack is empty. In each iteration of the while loop, we handle a single node - this is equivalent to a given function call in the recursive implementation. All the logic that is done in the function should be done in the while loop, including handling the children, which is done by pushing to the stack.

The time and space complexity of tree questions is usually straightforward. The time complexity is almost always $O(n)$, where n is the total number of nodes, because each node is only visited once, and at each node, $O(1)$ work is done. If more than $O(1)$ word is done at each node, let's say $O(k)$ work, then the time complexity will be $O(nk)$.

For space complexity, even if you are using recursion, the calls are still placed on the call stack which counts as extra space. The largest the stack will be (for either iterative or recursive) at any time will depend on the tree. For recursion, in the worst case it is $O(n)$ if the tree is just a straight line, so usually, the correct answer to give for space complexity is $O(n)$. If the tree is "complete" (all nodes have 0 or 2 children and each level except the last if full), then the space complexity is $O(logn)$ (the height of the tree is logarithmic in the number of nodes), but this is a best-case scenario.

Important note regarding iterative implementations: in the coded, we are adding `node.left` before `node.right`. Popping from a stack removes the most recently added element, thus we are actually visiting the right subtree first in the above code. In the recursive implementation, we visit the left subtree first. It doesn't matter in this problem because the only thing that matters is

that we visit all the nodes, regardless of order. However, it is still good to understand that when working iteratively, the visit order is opposite the insertion order.

Example 2: 112 - Path Sum

Given the `root` of a binary tree and an integer `targetSum`, return `true` if there exists a path from the root to a leaf such that the sum of the nodes on the path is equal to `targetSum`, and return `false` otherwise.

First, what information do we need at each function call? We need the current node, but what else? If we also keep an integer `curr` that represents the current sum of the nodes from the root to the current node, we can check this value against `targetSum` when we find a leaf. Thus, let's have a helper function `dfs(node, curr)` that returns `true` if there is a path starting at `node` and ending at a leaf with a sum equal to `targetSum`, if we already have `curr` contributed towards the sum.

What are the base cases? First, if we have an empty tree, we can't have a path as there are no nodes, so return `false`. If we are at a leaf node (which we can check by seeing if both children are `null`), then return `(curr + node.val) == targetSum`.

Otherwise, if we are not at a leaf, we could either continue down the left path or the right path. We only need one path to equal `targetSum`, so return `true` if **either** works. Don't forget to add the current node's value to `curr`.

```
class Solution:
    def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
        def dfs(node, curr):
            if not node:
                return False

            # if both children are null, then the node is a leaf
            if node.left == None and node.right == None:
                return (curr + node.val) == targetSum

            curr += node.val
            left = dfs(node.left, curr)
            right = dfs(node.right, curr)
            return left or right
```

```
        return dfs(root, 0)
```

Here's the iterative approach (remember that the iterative approach is much less common and should only be used if an interviewer asks for it):

```
class Solution:
    def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
        if not root:
            return False

        stack = [(root, 0)]
        while stack:
            node, curr = stack.pop()
            # if both children are null, then the node is a leaf
            if node.left == None and node.right == None:
                if (curr + node.val) == targetSum:
                    return True

            curr += node.val
            if node.left:
                stack.append((node.left, curr))
            if node.right:
                stack.append((node.right, curr))

        return False
```

Again, the time and space complexity are both $O(n)$, where n is the number of nodes in the tree, as each node is visited at most once and each visit involves constant work. In the worst case scenario for space (a straight line like a linked list), the recursion call stack will grow to the same size as the number of nodes in the tree.

Example 3: 1448 - Count Good Nodes in Binary Tree

Given the `root` of a binary tree, find the number of nodes that are **good**. A node is good if the path between the root and the node has no nodes with a greater value.

Again, let's start by thinking about what information we need at each function call (other than the node). At each node, we want to know if the node is good, and to know if the node is good,

we need to know the largest value between the root and the node. Let's use an integer `maxSoFar` to store this.

Then, we can use a function `dfs(node, maxSoFar)` that returns the number of good nodes in the **subtree rooted at `node`**, where the maximum number seen so far is `maxSoFar`.

What is the base case? If we have an empty tree, then the answer is `0` because there are no nodes, so there are no good nodes.

The total good nodes in a subtree is the number of good nodes in the left subtree + the number of good nodes in the right subtree + 1 if the current node is a good node. If `node.val >= maxSoFar`, that means the current node is a good node. Then we also need to find how many good nodes are in the left and right subtrees, which we can do by making recursive calls while updating `maxSoFar`.

```
class Solution:
    def goodNodes(self, root: TreeNode) -> int:
        def dfs(node, max_so_far):
            if not node:
                return 0

            left = dfs(node.left, max(max_so_far, node.val))
            right = dfs(node.right, max(max_so_far, node.val))
            ans = left + right
            if node.val >= max_so_far:
                ans += 1

            return ans

        return dfs(root, float("-inf"))
```

Here's the iterative approach:

```
class Solution:
    def goodNodes(self, root: TreeNode) -> int:
        if not root:
            return 0

        stack = [(root, float("-inf"))]
        ans = 0
```

```
        while stack:
            node, max_so_far = stack.pop()
            if node.val >= max_so_far:
                ans += 1

            if node.left:
                stack.append((node.left, max(max_so_far, node.val)))
            if node.right:
                stack.append((node.right, max(max_so_far, node.val)))

        return ans
```

The time and space complexities are both $O(n)$ as usual.

Example 4: 100 - Same Tree

Given the roots of two binary trees `p` and `q`, check if they are the same tree. Two binary trees are the same tree if they are structurally identical and the nodes have the same values.

If `p` and `q` are the same tree, then the following are true:

1. `p.val = q.val`
2. `p.left` and `q.left` are the same tree
3. `p.right` and `q.right` are the same tree

The main idea is that if any two trees are the same, then their subtrees must also be the same.

The following condition can be used to check if `p` and `q` are the same tree:

`p.val == q.val && isSameTree(p.left, q.left) && isSameTree(p.right, q.right)`

We need base cases so that the recursion eventually terminates. If `p` and `q` are both `null`, then we can `return true`, because they are technically both the same (empty) tree. If either `p` or `q` is `null` but not the other, we should `return false`, as they are clearly not the same tree.

A good way to think about base cases is to think about a tree with only one node. Let's say that `p` and `q` are both one-node trees with the same value. The first boolean check `p.val == q.val` passes, so now we check the subtrees. Because the nodes don't have children, both calls to the left and right subtrees will trigger the base case and return true.

This is the beauty of recursion - you can trust it to terminate correctly if you inductively craft a solution.

```
class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        if p == None and q == None:
            return True

        if p == None or q == None:
            return False

        if p.val != q.val:
            return False

        left = self.isSameTree(p.left, q.left)
        right = self.isSameTree(p.right, q.right)
        return left and right
```

Here's an iterative approach. We are using the exact same code to check the cases. Instead of returning true during the traversal, we return false if any condition is broken, and then return true at the end if we managed to get through the trees without returning false.

```
class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        stack = [(p, q)]
        while stack:
            p, q = stack.pop()
            if p == None and q == None:
                continue

            if p == None or q == None:
                return False

            if p.val != q.val:
                return False

            stack.append((p.left, q.left))
            stack.append((p.right, q.right))
```

```
        return True
```

The time and space complexity are $O(n)$ again.

Bonus example: 236 - Lowest Common Ancestor of a Binary Tree

Given the `root` of a binary tree and two nodes `p` and `q` that are in the tree, return the **lowest common ancestor (LCA)** of the two nodes. The LCA is the lowest node in the tree that has both `p` and `q` as descendants (note: a node is a descendant of itself).

Again, we want our recursive function to return the answer to the question. What is the base case? If we have an empty tree, then no LCA exists - return `null`.

Otherwisem how can we tell if a node is the LCA? Let's say we are at the root, then there are 3 possibilities.

1. The root node is `p` or `q`. The answer **cannot** be below the root node, because then it would be missing the root (which is either `p` or `q`) as a descendant.
2. One of `p` or `q` is in the left subtree, and the other one is in the right subtree. The root must be the answer because it is the connection point between the two subtrees, and thus the lowest node to have both `p` and `q` as descendants.
3. Both `p` and `q` are in one of the subtrees. In that case, the root is not the answer because we could look inside the subtree and find a "lower" node.

Remember: because of the recursive nature of trees, we can translate the cases into and algorithm. We just need to figure out how to find the answer if it is the first or third case.

In the first case, if we see that the current node is either `p` or `q`, we don't need to worry about the subtrees at all, because we know the answer cannot be in them. Therefore, we can return something (non-null) right away. In the base case, we return `null`. Therefore, a call to a subtree returns a non-null value only if one of `p` or `q` is in that subtree. We should return `null` for a subtree that contains neither `p` or `q`.

Then, the second case is implied if both calls to the left and right subtrees return something non-null, and the third case is implied if only one of the calls returns something.

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q:
'TreeNode') -> 'TreeNode':
```

```
        if not root:
            return None

        # first case
        if root == p or root == q:
            return root

        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)
        # second case
        if left and right:
            return root

        # third case
        if left:
            return left

        return right
```

The time and space complexities are $O(n)$.

SEE PROBLEMS 11, 1026, 543